ICOT Technical Memorandom: TM-0952

TM-0952

Cu-Prolog 第二版処理系仕様書

津田　宏，橋田　浩一，安川　秀樹，
白井　英俊（中京大学）

September, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

『cu-Prolog 第二版 処理系仕様書』

津田宏 (ICOT第三研究室)
橋田浩一 (ICOT第三研究室)
安川秀樹 (ICOT第三研究室)
白井英俊 (中京大学)

<< 目次 >>

以上

<< cu-Prolog V2ユーザーズマニュアル >>

ICOT第3研究室 津田 宏

cu-Prolog

1990.9.1

これは制約論理型プログラミング言語cu-Prologのユーザーズマニュアルである。cu-Prologは、prologの素式列を制約にできるので、記号的、組み合わせ的な制約を自然に記述する事ができるという特徴がある。自然言語処理ならびにAI全般への応用に適するプログラミング言語である。

{目次}

---

[0] コンパイルの方法

cu-Prologは、C言語による以下のモジュールからなる。

- ヘッダファイル(構造体、変数定義):
  include.h, funclist.h, varset.h, globalv.h, sysp.h
- Prolog関係のプログラム:
  main.c, mainsub.c, new.c, read.c, print.c, refute.c, unify.c
- 組み込み述語関係のプログラム:
  defsysp.c, syspred1.c, syspred2.c, jpsgsub.c
- 制約変換関係のプログラム:
  cunify.c, genfunc.c, modular.c, split.c, reduce.c, integ.c

これらのモジュールをコンパイル、リンクすれば実行ファイルが得られる。
(例えばUNIXの場合は上記のファイルを一つのディレクトリに置き、

```
cc -o cuprolog *.c <cr>
```

とする。またはソフトにはmakefileも添付されているので

```
make <cr>
```

としてもよい。)

ただし、使用するOSやコンパイラによっては、コンパイル前にinclude.h中の定数を以下のように書き換える必要がある。

[0.1] #define MSDOS ????

(1) MS-DOSのCコンパイラでSmallモデルの場合
(intが16ビット、ポインタサイズが16ビット)
例: MS-C ver.4, Turbo-C ver.1.5

```
#define MSDOS 1
```

(2) MS-DOSのCコンパイラでLarge(Huge)モデルの場合
例: MS-C ver.4 (large), Turbo-C ver.1.5(huge)
(intが16ビット、ポインタサイズが32ビット)

```
#define MSDOS 2
```

(3) UNIXの場合 (デフォルト)
例: Sun-3, Symmetry
(int、ポインタサイズ共に32ビット)

```
#define MSDOS 0
```

[0.2] #define CPUTIME ???? 文
(1) UNIX4.2/4.3 BSDでtimes()がサポートされていて、1/n単位でCPU-timeを計測する時には

```
#define CPUTIME n      (デフォルトは60)
```

とする。

(2)Sun-4では

```
#define CPUTIME 1000000
```

とする。この場合にはclock()を用いて時間を計測する。

(3)それ以外では

```
#define CPUTIME 0
```

とする。この場合は、実行時にCPUタイムが表示されない。

[0.3] その他
cu-Prologには次の3つのデータ領域がある。
システムヒープ :プログラム節、新述語定義等
ユーザヒープ :探索時の一時的な構造等
ユーザスタック :探索時のポインタのつけかえ等
cu-Prologを実行中に、
user heap overflow が頻繁に起これば、 HEAP_SIZE
user stack overflow が頻繁に起これば、 USTACK_SIZE
system heap overflowが頻繁に起これば、SHEAP_SIZE
の値をそれぞれ増やし、コンパイルし直すとよい。

---

[1] 起動法(実行ファイルの名称はcuprologとする)
OSのプロンプトから

```
cuprolog <RETURN>
```

とする。また起動と同時に初期プログラムを読み込む場合には

```
cuprolog ファイル名 <RETURN>
```

とする。

```
***** cu - Prolog  Ver. 3.02 *****
Copyright: Institute for New Generation Computer Technology, Japan 1989
        help -> %h
```

-

のように、オープニングとcu-Prologのトップレベルのプロンプト(_)が表示される。

---

[2] 終了法
cu-Prologトップレベルで、

```
%^ <RETURN>
```

または

```
:-halt. <RETURN>
```

---

[3] プログラムの記述
[3.1] 用語の説明
項: アトム、変数、複合項から成る。

アトム: (1) 定数: 英小文字ではじまる文字列、又は'(シングルクォーテーション)で囲まれた任意の文字列。現バージョンでは、漢字は扱えない。
(2) ストリング: "(ダブルクォーテーション)で囲まれた任意の文字列
(3) 数値: 整数または浮動小数点実数

変数: 英大文字または_から始まる文字列。_のみでは無名変数で、任意の2つの無名変数は異なった変数として扱う。

複合項: pを文字列、t1,t2,...,tnを項としたとき、p(t1,t2,...,tn)を複合項という。n=0のときはpのみとなる(例えば、カット,fail,定数)。
他の項の引数となりうる複合項を関数と呼び、その時のpをファンクタと呼ぶ。
他の引数とならない複合項を述語と呼び、その時のpを述語名と呼ぶ。

[3.2] プログラム節の記述:
cu-Prologのプログラム節は,Constraint Added Horn Clause(CAHC: 制約付ホーン節)と呼ばれ、通常のホーン節に制約を加えたものになっている。CAHCは以下の3種類の節からなる。

(1). <事実>

```
A.
A ; C1,...,Cn.
```

(例)

```
fly(bird).
member(X,[X|Y]).
f(a,X,Y);c0(X),c1(Y).
```

(2). <規則>

```
A :- B1,...,Bm.
A :- B1,...,Bm;C1,...,Cn.
```

(m = 0 の時は事実節と同値である。)
(例)

```
bird(X):-fly(X).
member(X,[Y|Z]):-member(X,Z).
```

```
f(a,b,X,Y):-g(u,X),h(Y,X);c0(X,Y).
```

(3). <質問>

```
        :- B1,...,Bm.
        :- B1,...,Bm;C1,...,Cn.
```

(例)

```
:-fly(chiken).
:-member(X,[a,b,c]).
:-f(X,Y,a);c0(X,Y).
```

Aをヘッド(頭部)、B1,...,Bmをボディ(本体)、C1,...,Cmを制約と呼ぶ。

なお、Ver2.4から、素式そのものを変数として記述できるようになった。従って

```
        call(X):-X.
        not(X) :- X,!,fail.
        not(_).
```

により、call/1, not/1が定義できる。

[3.3]制約の記述

制約は以下に示すモジュラーという標準形でなければならない。

<定義(モジュラー)>

次の3条件を満たす時,C1,...,Cm はモジュラーである(nilもモジュラーとする).

1. Ciはモジュラー定義述語
2. Ciの引数は全て変数
3. C1,...,Cmで,同じ変数は2個所に現れない.

<定義(モジュラー定義述語)>

fがモジュラー定義述語とは,fの定義節の本体が全てモジュラーである事をいう。

(現システムでは、モジュラー定義のチェックは行っていない)

[3.4] リスト

特別なファンクタにリストがある。引数は2つである。

cons(A,B) を [A|B] と記述し、nil を □ と記述する。

以下のような略記もできる。

```
        [A|□]     --> [A]
        [A|[B]]   --> [A,B]
        [A|[B,C]] --> [A,B,C]
        [A,[B|C]] --> [A,B|C]
```

---

[4] コマンド一覧

cu-Prologのトップレベルのコマンドには以下のものが用意されている。

[4.1]Prologに関するコマンド

%h　　ヘルプの表示

%d 述語名

%d 述語名/arity　　述語の定義の表示

%d.　　簡約化により消された述語を除いた全ての述語の定義を表示する

%d/　　簡約化により消された述語を含め全ての述語の定義を表示する

%d?　　述語の名前と各種フラグの状況を表示する。最初に組込み述語が表示

される。
再帰的(バックトラックを行なう)述語 +
ファンクタ名 ^
のマークがつく。
次にユーザ述語が表示される
スパイフラグが設定されている述語 *
簡約化されている述語 -
再帰を含む述語 +
制約変換で作られた新述語 #
のマークがつく。

%f システムヒープの使用状況を表示する
%Q cu-Prologを終了する
%G 静的ガーベジコレクションを行う。プログラム節の内容を一時ファイル(TEMPF.###)に書き出して再度読み込む
%c 数 Prologの推論段数の最大値を設定する。この段数を越えたゴールは全て失敗と見なす。

[4.2]ファイル入出力に関するコマンド([6]を参照)

"ファイル名" プログラムファイルをエコーバックなしで読み込む
"ファイル名? プログラムファイルをエコーバックつきで読み込む
%l ファイル名 ログファイルを設定する
%l no ログファイルへの記録を中止する
%w ファイル名 プログラム節をファイルに書き出す

[4.3]デバッグに関するコマンド([8]を参照)

%p 述語名
%p 述語名/Arity 述語のスパイフラグを設定/解除するスイッチ
%p* 全述語のスパイフラグを設定する
%p. 全述語のスパイフラグを解消する
%p> 制約変換にスパイフラグを設定/解除するスイッチ
%p? スパイフラグが設定されている述語名を表示する
%s ステップトレースモードon/offのスイッチ
%t ノーマルトレースモードon/offのスイッチ

[4.4]制約変換に関するコマンド([7]を参照)

%l. 制約変換で作られた新述語の導入定義節を表示する
%a 制約変換を全モジュラーモードにする
%o 制約変換をM-Solvableモードにする
%r 制約変換で簡約化を行なうかどうかのスイッチ
%n 文字列 制約変換で作られる新述語名を設定する(デフォルトは"c")。
%M 数 制約変換の深さを設定する。初期値は50

[4.5]その他のコマンド

%C パーザ用組込みファンクタcat()を再定義する。[5.3]参照

---

[5]組込み述語、ファンクタ

[5.1]一般的な組込み述語

cu-Prolog Ver3.2では、以下の組込み述語が用意されている。
<1>は関数的なType1の述語、<2>は複数解を持つType2の述語を示す。

また、引数のモードで+は具体化された項、-はfree変数を表す。

!/0 :カット <2>
abolish/2 : <1> abolish(F,A) 述語F/Aの定義を消去する
assert/1,/2,/3: <1>
asserta/1,/2,/3: <1>
assertz/1,/2,/3: <1>
assert(a(X,Y),[b(X),c(Y)],[d(Y)])
head body constraint
により、a(X,Y):-b(X),c(Y);d(Y). が定義される
clause/3: clause(T+,B,C) 表式Tと単一化する頭部を持つ節の本体がB、制約がCである<2>
close/1: close(F) I/Oのために開いているファイルFを閉じる
concat/3 : <1> 文字列の連結 concat("ab","cd",X) => X="abcd"
concat2/2 : <1> concat("abc",X) => X=["a","b","c"]
count/1: count(C-) 呼び出し毎に異なる数を返す <1>
equal/2: 第1、第2引数を単一化する <1>
eq/2 : <1> eq(X,Y) XとYが等しいかチェックする
execute/1: execute([memb(X,[a,b,c]),memb(Y,[b,c,d])]) 引数のリストの中を順時実行する。
fail/0: 常に失敗する述語 <1>
functor/3: functor(H+,F,A) 表式Hの述語名がF、アリティがA <1>
gensym/1: gensym(C-) 呼び出しの度に異なるアトムを返す
geq/2: 数値比較関数 geq(X,Y) X>=Y
greater/2: 数値比較関数 greater(X,Y) X>Y
halt/0: <1> cu-Prologを終了する
leq/2: 数値比較関数 leq(X,Y) X<=Y
less/2: 数値比較関数 less(X,Y) X<Y
ml/2: <1> 複合項Tをリスト形式Lにする
memb/2: memb(A,L+) 組込み版member述語 <2>
multiply/3: multiply(X,Y,Z) X*Y=Z (引数が2つ以上具体化されている時)
name/2: name(X,L) 項Xの文字列のキャラクタがリストL <1>
nl/0: nl <1> 改行
nl/1
open/3
or/2,3,4,5
read/1,2
retract/1,/2,/3: <1> retract(Head,Body,Constraint)
引数と単一化するプログラム節を消去する。
see/1: see(F) ファイルFをcurrent input streamにする
seen/0: current input streamを閉じる
strlen/2: <1> strlen(S,L) 文字列Sの長さがL
sum/3: sum(X,Y,Z) X+Y=Z (引数のうち2つ以上具体化されているときのみ) <1>
tab/0: tab タブをプリントする <1>
tell/1: tell(F) ファイルFをcurrent output streamにする
told/0: current output streamを閉じる
true/0: 常に成功する述語。
var/1: 項がfree varのときTRUE <1>
write/1: 項をコンソールに表示する。ストリングは"を表示しない。<1>

[5.2] 制約変換に関する組込み述語
condname/2: condname([c0(X,Y),c1(Y,Z)],X) => X=[c0,c1]
リスト形式の制約を簡略化して表示する

pcon/0: この述語が実行された時点での制約を表示する <1>

unify(+C,-NC) : リストの形の制約Cをモジュラーに変換したものがNCである。
Cはリストの形の制約(例: [member(X,Y),append(X,Y,Z)])、
NCには変数を入れて呼ぶ。
成功すると、Cと等価でモジュラーな制約がNCに入る。

[5.3] JPSGパーザに関する組み込み述語、ファンクタ
cat/6 : 句構造のカテゴリーを表すファンクタ。
※cat()は、%Cコマンドでトップレベルから変更可能である。
%C [素性名1,素性タイプ1,素性名2,素性タイプ2,...]
と素性名と素性タイプをリストで囲んで指定する。
(注1)素性名は大文字から始まり5文字以内でなければならない。
(注2)素性タイプは、1,2,3の数値で指定する。ここで、

1 =	Normal	下の2つ以外
2 =	CatSingle	カテゴリを一つ取る素性(Adjacent等)
3 =	CatSet	カテゴリの集合を取る素性(Subcat等)

である。
デフォルトでは、[POS,1,FORM,1,AJA,2,AJN,2,SC,3,SEM,1] つまり

素性名	素性タイプ	対応するJPSGの素性
POS	Normal	pos
FORM	Normal	gr, vform, pform, etc.
AJA	CatSingle	ajacent
AJN	CatSingle	ajunct
SC	CatSet	subcat
SEM	Normal	sem

となっている。

t(M,L,R):ヒストリーを表すファンクタ。パーザーで木構造を表示するときに
履歴を格納する。
<定義(ヒストリー)>
(1)カテゴリーはヒストリー
(2)C, Wがカテゴリーの時、t(C,W,□) はヒストリー
(3)L, Rがヒストリーで、
Mがカテゴリー又はt(C,W,□) の形(C,Wはカテゴリー)の時、
t(M,L,R)はヒストリー

tree(H):ヒストリーHを木構造でプリントする述語。
H = t(C,W,□) の時 C--W
H = t(M,L,R) の時

```
---M
   |
   |--L
   |
   |--R
```

と木構造で表示をする。

---

[6] ファイルの入出力

[6.1] プログラムの読み込み

システム起動と同時に読み込むには、OSのプロンプトから

cuprolog ファイル名 <RETURN>

とする。エコーバックはない。

トップレベルからエコーバックせずに読み込む:

"ファイル名" <RETURN>

トップレベルからエコーバックありで読み込む(デバッグ時など):

"ファイル名? <RETURN>

[6.2] プログラムの保存

%w ファイル名

[6.3] ログファイルの設定、解除

%l ログファイル名 <RETURN>

によりログファイルを設定すると、以後の画面がファイルに格納される。

ログを中止するには

%l no <RETURN>

---

[7] 制約変換機構

cu-Prologでは制約変換機構のみを単独で使うこともできる。

[7.1] @モード:トップレベルから制約変換機構を単独で使う。

@ <制約の並び>. <RETURN>

とする。制約と等価でモジュラーな制約とそれらの定義を返す。

(例): @ member(X,[a,b,c,d,e]),member(X,[b,n,f]).

[7.2] unify(C,NC): 組込み述語([5.2])

Prologの中で変換ルーチンを使う。

[7.3] 制約変換に関するコマンド

制約変換を制御するコマンドには以下が用意されている

%r 新述語定義節の簡約化on/offのスイッチ。デフォルトはon。

%n <述語名> 制約変換で作られる新述語名の最初の部分を設定する。デフォルトは"c"で、c0,c1,...という述語が定義される。

%a 全モジュラーモード。新述語の定義はすべてモジュラーになる。デフォルトはこちらのモード。

%o m-solvableモード。新しく定義される述語の定義のうち、一つの節だけモジュラーにする。

%L 制約変換でできた新述語の導入節を表示する。

---

[8] トレース機能

cu-Prologには、デバッグ用にトレース機能がある。

まず最初に述語にスパイフラグを設定してから、次のいずれかのトレースモードを選ぶ。

(1)ノーマルトレース……(プロンプトは$になる)途中でストップしない

(2)ステップトレース……(プロンプトは>になる)実行を一時停止し、入力待ちになる。リターンで実行を継続、sでゴールの最左素式が(TRUEまたはFALSEに)解かれるまで表示を中断、zでcu-Prologトップレベルに戻る。
以降トレースモードを解除するまで、スパイフラグのついた述語を含む素式が一番左に現れているゴールが表示される。

[8.1] スパイフラグの設定
スパイされていない述語にスパイフラグを設定する:
%p 述語名 <RETURN>
%pコマンドはスイッチになっているので、既にスパイされている述語に再び%pを実行すると、述語のスパイフラグは解消される。
全述語にスパイフラグを設定する:
%p* <RETURN>
全述語のスパイフラグを解消する:
%p. <RETURN>
スパイフラグが設定されている述語名を表示する:
%p? <RETURN>
制約変換ルーチン(fold/unfold変換)にスパイフラグを設定/解消する:
%p> <RETURN>

[8.2] ステップトレースモード
cu-Prologトップレベルから
%s <RETURN>
とすると、ステップトレースモードとなり、カーソルの形状が>になる。%sはスイッチになっており、再度実行すると通常モード(カーソルは_)になる。
ステップトレースモードでは、スパイフラグが設定された述語が一番左に現れるゴールの実行の手前で一時停止し入力待ちとなり、
リターン : continue
s : skip (一番左のゴールが解かれるまで表示を中断)
z : quit refutation
をユーザが指定できる。

[8.3] ノーマルトレースの設定
cu-Prologトップレベルから
%t <RETURN>
とするとノーマルトレースモードとなり、カーソルの形状が$になる。%tはスイッチになっており、再度実行すると通常モードになる。
ノーマルトレースモードでは、スパイフラグが設定された述語が一番左に現れるゴールを表示する。

---

[9] 参考:BNF記法によるcu-Prologの構文

```
<alpha numerical char>::=<capital> | <small> | <digit>
<capital>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<small>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>::=0|1|2|3|4|5|6|7|8|9
<name>::=<alpha numerical char> | <alpha numerical char><name>
<string>::=<empty> | <name>
<var>::=<capital><string> | _<string>
<constant>::=<small><string> | '<string>' | <number> | "<string>"
```

```
<functor>::=<name>
<predicate>::=<name>
<term>::=<var> | <constant> | <functor>(<term list>)
<term list>::=<term> | <term>,<term list>
<atomic formula>::=<predicate>(<term list>) | <predicate>
<atomic formula list>::=<atomic formula> |
                        <atomic formula>,<atomic formula list>
<constraint>::=<atomic formula list>
<clause>::=
        <atomic formula>. |
        <atomic formula>;<constraint>. |
        <atomic formula>:-<atomic formula list>. |
        <atomic formula>:-<atomic formula list>;<constraint>. |
        ?-<atomic formula list>.|
        ?-<atomic formula list>;<constraint>.
```

---

[10] モデルセッション

◎ セッション1: 制約変換ルーチンを動かす

```
tsuda#icot21[4]% cuprolog   % cu-Prologを起動する

 *****  cu - Prolog   Ver. 3.00  *****
        (c) ICOT, H.Tsuda and K. Hasida
          help ->  %h

_member(X,[X|Y]).                % member/2述語の定義
_member(X,[Y|Z]):-member(X,Z).
_append([],X,X).                 % append/3述語の定義
_append([A|X],Y,[A|Z]):-append(X,Y,Z).

@ member(X,[a,b,c]).             % member(X,[a,b,c])を等価な制約に変換

solution = c0(X)
c1(b).
c1(c).
c0(a).
c0(X0):-c1(X0).
CPU time = 0.017 sec

@ member(X,[a,b,c,d,e]),member(X,[c,d,e,f,g]).  % 変換

solution = c3(X)
c6(d).
c6(e).
c5(c).
c5(X0):-c6(X0).
c3(X0):-c5(X0).
CPU time = 0.050 sec
```

```
© member(A,X),append(X,Y,Z).    % fold 変換を用いる例

solution = c8(A, X, Y, Z)
c8(X2, [X2|Y3], Y0, [X2|Z1]):-append(Y3, Y0, Z1).
CPU time = 0.017 sec

%L      % 変換の新述語定義のリスト
 +-- List integrate() traces --+
<5,2> c9(A0, A1, X2, Y3, Z4) <=> append(X2, Y3, Z4), member(A0, [A1|X2]).
<4,2> c8(A, X, Y, Z) <=> append(X, Y, Z), member(A, X).
<1,2> c7(X0) <=> member(X0, [e]), member(X0, [c,d,e,f,g]).
<1,2> c6(X0) <=> member(X0, [d,e]), member(X0, [c,d,e,f,g]).
<1,2> c5(X0) <=> member(X0, [c,d,e,f,g]), member(X0, [c,d,e]).
<1,2> c4(X0) <=> member(X0, [c,d,e,f,g]), member(X0, [b,c,d,e]).
<1,2> c3(X) <=> member(X, [c,d,e,f,g]), member(X, [a,b,c,d,e]).
<1,1> c2(X0) <=> member(X0, [c]).
<1,1> c1(X0) <=> member(X0, [b,c]).
<1,1> c0(X) <=> member(X, [a,b,c]).

%p+++ spy member/2       % member/2述語をスパイする

%t                       % ノーマルトレースモードにする
 +++ normal trace on +++

:-member(X,[b,c,d]).     % Prologの質問節
 [0>>member(X_0, [b,c,d])
   <=0-1=member(X, [X|Y]).
success.
member(b, [b,c,d])
  X = b;                 % 別解を探索する時は;を入力
 [0>>member(X_0, [b,c,d])
   <=0-2=member(X, [Y|Z]):-member(X, Z).
 [1>>member(X_26, [c,d])
   <=1-1=member(X, [X|Y]).
success.
member(c, [b,c,d])
  X = c;                 % 別解
 [1>>member(X_26, [c,d])
   <=1-2=member(X, [Y|Z]):-member(X, Z).
 [2>>member(X_44, [d])
   <=2-1=member(X, [X|Y]).
success.
member(d, [b,c,d])
  X = d;                 % 別解
 [2>>member(X_44, [d])
   <=2-2=member(X, [Y|Z]):-member(X, Z).
 [3>>member(X_62, [])
   <=3-no= fail member.
no.
CPU time = 0.050 sec
```

```
%p>                          % 制約変換をスパイ
 +++ spy fold/unfold transformation

$%n new                      % 新述語の名前をnewから始まるものにする

$@ member(A,Z),append(X,Y,Z).   % 制約変換

1 transfer  member(A_6, Z_8), append(X_10, Y_12, Z_8)  : A_6, Z_8, X_10, Y_12
    <1> new predicate new10 is defined.
1=1(new10)  <<- append(□, X, X).
2 transfer  member(A_6, X_164)  : A_6, X_164
2 => member(A0, X1)
1=2(new10)  <<- append([A|X], Y, [A|Z]):-append(X, Y, Z).
2 transfer  append(X_166, Y_168, Z_170), member(A_6, [A_164|Z_170])  : A_6, A_164↓
↑, X_166, Y_168, Z_170
    <2> new predicate new11 is defined.
2=1(new11)  <<- member(X, [X|Y]).
3 transfer  append(X_166, Y_168, Y_323)  : X_166, Y_168, X_321, Y_323
3 => append(X0, Y1, Y3)
2=2(new11)  <<- member(X, [Y|Z]):-member(X, Z).
3 transfer  member(X_321, Z_325), append(X_166, Y_168, Z_325)  : X_166, Y_168, X_↓
↑321, Y_323, Z_325
3===fold new10(A, Z, X, Y) <=> append(X, Y, Z), member(A, Z)
3 => new10(X2, Z4, X0, Y1)
2 => new11(A0, A1, X2, Y3, Z4)
1 => new10(A, Z, X, Y)
solution = new10(A, Z, X, Y)
new11(X2, X2, X0, Y1, Y3):-append(X0, Y1, Y3).
new11(X2, Y3, X0, Y1, Z4):-new10(X2, Z4, X0, Y1).
new10(A0, X1, □, X1):-member(A0, X1).
new10(A0, [A1|Z4], [A1|X2], Y3):-new11(A0, A1, X2, Y3, Z4).
CPU time = 0.067 sec

$%Q     % cu-Prolog終了
```

◎ セッション2: 制約付ホーン節(CAHC)によるプログラム

```
_"t2.p?          % ユーザーバック付きでt2.pというファイルを読み込む
=== open 't2.p'
member(X,[X|Y]).          % member/2述語定義
member(X,[Y|Z]):-member(X,Z).
append(□,X,X).            % append/3述語定義
append([A|X],Y,[A|Z]):-append(X,Y,Z).
inter(A,X,Y):-member(A,X);member(A,Y). % CAHCによる共通要素の計算

:-inter(X,[p,o,i,p,o,i,p,o,i,p,o,i],[x,c,v,a,s,d,a,s,d,a,s,da]).
                % CAHCによる実行
no.
CPU time = 0.183 sec
```

```
:-member(X,[p,o,i,p,o,i,p,o,i,p,o,i]),member(X,[x,c,v,a,s,d,a,s,d,a,s,da]).
                % 通常のPROLOGによる実行
no.
CPU time = 0.200 sec

@member(X,[p,o,i,p,o,i,p,o,i,p,o,i]),member(X,[x,c,v,a,s,d,a,s,d,a,s,da]).
                % 制約変換のみによる実行
solution = fail.
CPU time = 0.217 sec
 ****** end of file *******
```

◎ セッション3: JPSGパーザの実行例

```
_"jpsg.p"               % JPSGパーザプログラムの読み込み
=== open 'jpsg.p'

 ****** end of file *******
CPU time = 0.917 sec
```

% 述語 p/1がトップレベルの呼び出しである。パーザーは引数で与えられた文の解析木と最上位ノードのカテゴリと制約を返す。

```
_:-p([ken,ga,naomi,wo,ai,suru]). % 「健が奈緒美を愛する」

v[syusi]:[love,ken,naomi]---[suff_p]   % 対応する解析木
 |
 |--v[vs2]:[love,ken,naomi]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p] % 親ノード(adjacent feature principle)
 |   |   |
 |   |   |--n[n]:ken---[ken]    % 右娘ノード(Head)
 |   |   |
 |   |   |__p[ga, AJA:n[n]]:ken---[ga] % 左娘ノード (Adjacent)
 |   |
 |   |__v[vs2, SC:{p[ga]}]:[love,ken,naomi]---[subcat_p]
 |       |
 |       |--p[wo]:naomi---[adjacent_p]
 |       |   |
 |       |   |--n[n]:naomi---[naomi]
 |       |   |
 |       |   |__p[wo, AJA:n[n]]:naomi---[wo]
 |       |
 |       |__v[vs2, SC:{p[wo], p[ga]}]:[love,ken,naomi]---[ai]
 |
 |__v[syusi, AJA:v[vs2]]:[love,ken,naomi]---[suru]

cat     cat(v, syusi, □, □, □, [love,ken,naomi])
cond                            % 曖昧でないから制約は空
true.
CPU time = 0.217 sec
```

```
_:-p([ken,ga,ai,suru]). %「健が愛する」
                        % 「健が(誰かを)愛する」「健が愛する(誰か)」
                        % の曖昧性あり
v[Form_865, AJN:Adj, SC:{SubCat}]:SEM_871---[suff_p]
 |
 |--v[vs2, SC:{p[wo]}]:[love,ken,Obj0_571]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p]
 |   |   |
 |   |   |--n[n]:ken---[ken]
 |   |   |
 |   |   |__p[ga, AJA:n[n]]:ken---[ga]
 |   |
 |   |__v[vs2, SC:{p[ga], p[wo]}]:[love,ken,Obj0_571]---[ai]
 |
 |__v[Form_865, AJA:v[vs2], AJN:Adj, SC:{SubCat}]:SEM_871---[suru]

cat     cat(v, Form_865, □, Adj_867, SubCat_869, SEM_871)
cond    c6(Form_865, SubCat_869, Obj0_571, Adj_867, SEM_871)
true.
CPU time = 0.150 sec
                % 制約c6(...)に曖昧性がパックされている。

_:-c6(F,SC,_,AJ,SEM).           % 制約c6(...)を解く
  F = syusi  SC = [cat(p, wo, □, □, □, Obj00_34)]  AJ = □  SEM =
[love,ken,Obj00_34];
                                % 「健が(誰かを)愛する」に対応する解
  F = rentai  SC = □  AJ = [cat(n, n, □, □, □, inst(Obj0_34,
Type1_36))]  SEM = inst(Obj0_34, [and,Type1_36,[love,ken,Obj0_34]]);
                                % 「健が愛する(誰か)」に対応する解
no.
CPU time = 0.017 sec
```

※※※※〔 注意事項 〕※※※※※※※※※※※※※※※※※※※※※※※※

cu-Prologは無保証なソフトウェアであり、本説明書の仕様は変更することがあります。cu-PrologはICOTのツール指定を受けているため、著作権はICOTに帰属します。cu-Prologは研究用にのみ使用して下さい。
ソフトウェアにバグを見つけられた方、及び御意見のある方は、下記まで御連絡下さい。バージョンアップの参考にさせていただきます。

[お問い合わせ先]
(財)新世代コンピュータ技術開発機構 第3研究室
津田 宏
住所 : 〒108 東京都港区三田1-4-28 三田国際ビル21F
TEL : (03)456-3069 FAX : (03)456-1618
E-mail: tsuda@icot.or.jp

A Guide to cu-PrologV2

TSUDA, Hiroshi
Third Research Laboratory
Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
E-mail: tsuda@icot.or.jp

August 27, 1990

Contents

1 Introduction

This is the user's manual of cu-PrologV2[5, 4].

cu-Prologis an experimental constraint logic programming language. Unlike most conventional CLP systems, cu-Prologallows user-defined predicates as constraints and is suitable for implementing a natural language processing system based on the unification-based grammar[2]. As an application of cu-Prolog, we developed a JPSG (Japanese Phrase Structure Grammar [1]) parser with the JPSG Working Group (the chairman is Prof. GUNJI, Takao of Osaka University) at ICOT. cu-Prologis also the complete implementation of the constraint unification[3] and its name (cu) comes from the technique. cu-Prologis implemented on UNIX in C language.

2 How to Compile cu-Prolog

The source codes of cu-Prologconsists of the following 5 header files and 14 program files.

- header files:

  ```
  include.h funclist.h varset.h globalv.h sysp.h
  ```

- program files concerning Prolog:

  ```
  main.c mainsub.c new.c read.c print.c refute.c unify.c
  ```

- program files concerning system built-in predicates:

  ```
  defsysp.c syspred1.c syspred2.c jpsgsub.c
  ```

- program files concerning constraint transformation:

  ```
  cunify.c genfunc.c modular.c split.c reduce.c integ.c
  ```

To get the execution code of cu-Prolog, you have only to compile and link all the modules (`*.c` files) as follows (the following is in UNIX case).

```
cc -o cuprolog *.c [CR]
```

or

```
make [CR]
```

Before compiling, you may have to rewrite some statements in **include.h** to your computer.

2.1 `#define MSDOS xxx`

1. For MS-DOS small model(int size is 16bit, pointer size is 16bit):

   ```
   #define MSDOS 1
   ```

2. For MS-DOS large(huge) model (int16bit, pointer32bit):

   ```
   #define MSDOS 2
   ```

3. For UNIX (int32bit, pointer32bit):

   ```
   #define MSDOS 0
   ```

2.2 #define CPUTIME xxx

1. If your C compiler has **times()** function, [1]

   ```
   #define CPUTIME 60
   ```

   If times() in your system returns CPU time in N-th of a second,

   ```
   #define CPUTIME N
   ```

2. For Sun-4 system, clock() function is used on behalf of times().

   ```
   #define CPUTIME 1000000
   ```

3. Otherwise, the CPU time is not printed.

   ```
   #define CPUTIME 0
   ```

2.3 Other #define statements.

In executing cu-Prolog,

- If "**user heap overflow**" error occurs frequently, increase the value of **HEAP_SIZE**.
- If "**user stack overflow**" error occurs frequently, increase the value of **USTACK_SIZE**.
- If "**system heap overflow**" error occurs frequently, increase the value of **SHEAP_SIZE**.

3 How to start and quit cu-Prolog

To start cu-Prolog, type the following in OS.

```
cuprolog [CR]
```

To start cu-Prologwith reading an initial program, type

```
cuprolog filename [CR]
```

To quit cu-Prolog, type followings at the top level of cu-Prolog.

```
%Q [CR]
```

or

```
:-halt.  [CR]
```

4 Syntax of cu-Prolog

term : atom, variable, complex term

atom : the string that begins with a small letter.

variable : the string that begins with a capital letter.

complex term : let p be a string and $t_1, t_2, \cdots, t_n$ be terms, then $p(t_1, t_2, \cdots, t_n)$ be a complex term. p is called a *functor* or a *predicate symbol*. List is a special functor.

[1] **times()** is the UNIX 4.2/3 BSD Library which returns CPU time in 60th of a second.

4.1 Constraint Added Horn Clause (CAHC)

The program clauses of cu-Prologis called Constraint Added Horn Clause(CAHC) and has the following forms:

1. Fact

$$H.$$
$$H : C_1, \cdots, C_n.$$

2. Rule

$$H :- B_1, \cdots, B_m.$$
$$H :- B_1, \cdots, B_m ; C_1, \cdots, C_n.$$

3. Question

$$:- B_1, \cdots, B_n.$$
$$:- B_1, \cdots, B_n ; C_1, \cdots, C_n.$$

$H, B_1, \cdots, B_n$, and $C_1, \cdots, C_n$ are called Head, Body, and Constraint respectively.

cu Prologallows a variable as an atomic formula. By the following programs, `call/1` and `not/1` are defined.

```
call(X) :- X.
not(X)  :- X, !, fail.
not(_).
```

4.2 Canonical form of Constraint

In CAHC, Constraint must be a canonical form called **modular**.

[Def] 1 (modular) *A sequence of atomic formulas $C_1, C_2, \ldots, C_m$ is* modular *when*

1. *every argument of C_i is a variable $(1 \leq i \leq m)$, and*
2. *no variable occurs in two distinct places, and*
3. *the predicates occurring in C_i are modularly defined $(1 \leq i \leq m)$.*

The predicate occurring in the constraint of CAHC is an ordinary Prolog predicate of the following form.

[Def] 2 (modularly defined) *Predicate* p *is modularly defined, when in its every definition clause of the form, $P :- \mathbf{D}.$,*

D *is modular or empty.*

4.3 BNF description of cu-Prolog

Table 1 is the BNF description of the syntax of cu-Prolog.

5 Summary of system commands

This section lists all the system commands from the top level of cu-Prolog. *predicate* represents *predicate_name* or *predicate_name/arity*.

$$
\begin{aligned}
< char > &::= < capital > | < small > | < digit > \\
< capital > &::= A|B|C|\ldots|X|Y|Z \\
< small > &::= a|b|c|\ldots|x|y|z \\
< digit > &::= 0|1|2|3|4|5|6|7|8|9 \\
< series > &::= < digit > | < digit >< series > \\
< number > &::= < series > | < series > . < series > \\
< name > &::= < char > | < char >< name > \\
< string > &::= < empty > | < name > \\
< var > &::= < capital >< string > |\_ < string > \\
< constant > &::= < small >< string > |' < name >' | < number > \\
< functor > &::= < name > \\
< predicate > &::= < name > \\
< term > &::= < var > | < constant > | < functor > (< termlist >) \\
< termlist > &::= < term > | < term >, < termlist > \\
< af > &::= < predicate > (< termlist >)| < predicate > | < var > \\
< aflist > &::= < af > | < af >, < aflist > \\
< horn > &::= < af > | < af >: - < aflist > |?- < aflist > \\
< cahc > &::= < horn > .| < horn >;< aflist > .
\end{aligned}
$$

Table 1: BNF description of cu-Prolog

5.1 Prolog commands

%h	help
# *OS_command*	execute OS command.
%d *predicate*	list definition clauses of a predicate
%d.	list all
%d/	list all (include reduced predicates)
%d?	list all predicate names
	for system predicates, +:recursive, ~:functor
	for user predicates, *:spied, −:reduced, +:recursive,#: new predicates in constraint transformation
%f	show free system heap size
%Q	quit cu-Prolog
%G	(static) garbage collection
%c *number*	set maximum depth of resolution

5.2 File I/O commands

"*filename*"	consult file without echo back
"*filename*?	consult file with echo back
%l *filename*	set log file name
%l no	reset log file
%w *filename*	write the current program to file

5.3 Debug commands

%p *predicate*	switch (on/off) spy points on the predicate
%p *	set spy points on all the predicates
%p .	remove spy points on all the predicates
%p >	switch (set/remove) spy points on constraint transformation
%p ?	list spied predicates
%t	normal trace mode on/off
%s	step trace mode on/off

5.4 Constraint Transformation commands

%L		list the history of integrate() (in CT)
%a		all modular mode (in CT)
%o		M-Solvable mode (in CT)
%r		switch (on/off) reduction mode (in CT)
%n	*predicate*	set new predicates name (in CT) (c0,c1,...)
%M	*number*	set maximum depth of constraint transformation (default is 60)

5.5 Other commands

%C redefine cat() functor for JPSG parser

6 Built-in predicates, functors

6.1 General predicates

The following are general built-in predicates of cu-Prolog.

PREDICATE	MEANING
!/0	cut
abolish/2	abolish(F,A) delete def. clauses of a predicate F/A
assert/1,2,3	assert(Head, [Body],[Constraint])
asserta/1,2,3	assert a clause at the top.
assertz/1,2,3	assert a clause at the bottom.
clause/3	clause(T+,B,C) C is a constraint.
close/1	close(F) close file pointer F
concat/3	concatenate strings concat("ab","cd","abcd")
concat2/2	concat2("ab",["a","b"])
count/1	return different number (cf. gensym in LISP)
equal/2	equal(X,Y) unify X and Y
eq/2	eq(X,Y) check X is equal to Y
execute/1	execute([memb(X,[a,b]),memb(X,[b,c])]) execute goals given in list
fail/0	always fail
functor/3	functor(H+,F,A) predicate name of H is F/A
gensym/1	return different string (cf. count/1)
geq/2	geq(X,Y) check X>=Y
greater/2	greater(X,Y) check X > Y
halt/0	quit cu-Prolog
leq/2	leq(X,Y) check X=<Y
less/2	less(X,Y) check X < Y
ml/2	univ. ml(T,L) is T=..L
memb/2	builtin member
multiply/3	multiply(X,Y,Z) is X * Y = Z
name/2(A,L)	name(A,L) the name of the atom A is the list L
nl/0	write '\n'
open/3	
or/2,3,4,5	
read/1,2	
retract/1,2,3	retract(Head, [Body],[Constraint])

`see/1`	`see(F)` F becomes the current input stream
`seen/0`	close the current input stream
`strlen/2`	`strlen(S,L)` L is the length of string S
`sum/3`	`sum(X,Y,Z)` is `X + Y = Z`
`tab/1`	write '\t'
`tell/1`	`tell(T)` T becomes the current output stream
`told/0`	close the current output stream
`true/0`	always true
`var/1`	`var(T)` T is a free variable
`write/1`	`write(T)` write term T

6.2 Special built-in predicates for constraint transformation

The following are special built-in predicates for constraint transformation.

`condname/2:` make predicate name list `condname([c0(X,Y),c1(Y,Z)],[c0,c1])`

`pcon/0:` write all the remaining constraints.

`unify(C+,NC-):` `C` and `NC` are lists of atomic formulas and `NC` is a modular form of `C`.

6.3 Special built-in predicates for JPSG parser

The followings are special built-in predicates for JPSG parser.

`cat/6:` the functor representing the category of JPSG. See below.

`tree(H):` draw history `H` in tree format.

`t(M,L,R):` the functor representing a history.

`cat/6` functor is set by `%C` command from the top level of cu-Prologas follows.

```
%C [CatName1,CatType1,CatName2,CatType2,...]
```

Here, CatName is a feature name which begin with a capital letter and be shorter than 5letters. CatType is a feature type.

CatType		
1	Normal	other than 2,3
2	SingleCat	takes one category (such as Adjacent feature)
3	SetCat	takes set of categories (such as Subcat feature)

Default `cat` functor is defined as `[POS,1,FORM,1,AJA,2,AJN,2,SC,3,SEM,1]` that is,

CatName	feature type	JPSG features
POS	Normal	pos
FORM	Normal	gr,vform,pform, and so on
AJA	SingleCat	ajacent
AJN	SingleCat	ajunct
SC	SetCat	subcat
SEM	Normal	sem

[Def] 3 (history) *history is defined as follows.*

- *a category is a history*
- *if* `C` *and* `W` *are categories, then* `t(C,W,[ ])` *is a history.*
- *if* `L` *and* `R` *are histories and* `M` *is category, then* `t(M,L,R)` *is a history.*
- *if* `L` *and* `R` *are histories, and* `C` *and* `W` *are categories, then* `t(t(C,W,[ ]),L,R)` *is a history.*

`tree(t(C,W,[ ]))` writes

```
C--W
```

and `tree(t(M,L,R))` writes

```
---M
   |
   |-L
   |
   |-R
```

7 File I/O

7.1 Read a program

- Start cu-Prologfrom OS with reading an initial file:

  `cuprolog` *filename* `[CR]`

- Read a file in the top level of cu-Prologwithout echo back:

  "filename" `[CR]`

- Read a file in the top level of cu-Prologwith echo back:

  "filename? `[CR]`

7.2 Save a program

To save current program clauses to a file in the top level of cu-Prolog,

`%w` *filename* `[CR]`

7.3 Log file

- Set log file :

  `%l` *filename* `[CR]`

- End log file :

  `%l no [CR]`

8 Constraint Transformation

You can use the constraint transformation module of cu-Prologalone in two ways.

8.1 '@' command

First, from the top level of cu-Prolog, type '@' followed by a sequence of atomic formulas and a period. For example, by typing as follows,

```
@ member(X,[a,b,c]),member(X,[b,c,d]).  [CR]
```

Then, cu-Prologreturns the equivalent modularly defined constraint and its definitions.

8.2 unify/2 predicate

Second, use the constraint transformation routine as a Prolog procedure. cu-Prologhas the predicate **unify(OldCond, NewCond)**. Constraints are described by list as follows.

```
[c0(X,Y), c1(P,Q,R), c2(Q,S)]
```

This predicate succeeds if and only if **OldCond** has been instantiated to constraints and **NewCond** is a free variable. **NewCond** is instantiated to the modularly defined constraint that is equivalent to **OldCond**.

9 Program trace

9.1 Set spy points

%p *	set spy points on all the predicates.
%p .	remove spy points on all the predicates.
%p *predicate*	switch (set/remove) the spy point on the predicate
%p >	switch (set/remove) the spy point on the unfold/fold transformation.
%p ?	list spied predicates.

9.2 Set trace

Tracing is to show the calls and exits of spied predicates.

%s switch step (interactive) trace (on/off). In this mode, the prompt is '>'.
%t switch normal trace (on/off). In this mode, the prompt is '$'.

In the step trace mode, the execution stops and waits user's input.

[CR]	continue
s	skip tracing
z	quit refutation

10 Model session

This section shows a model session of cu-Prolog.

10.1 Constraint transformation and trace

```
tsuda#icot21[1] cuprolog                % start cu-prolog

 *****  cu - Prolog   Ver. 3.02  *****
 Copyright: Institute for New Generation Computer Technology, Japan 1989
           help ->  %h

_member(X,[X|Y]).                % define member()
_member(X,[Y|Z]):-member(X,Z).
_append([],X,X).                 % define append()
_append([A|X],Y,[A|Z]):-append(X,Y,Z).

_@ member(X,[a,b,c]).                % transform member(X,[a,b,c])

solution = c0(X)                 % returns modular form
```

```
c1(b).                          % new clauses (c0 and c1 are new predicates)
c1(c).
c0(a).
c0(X0):-c1(X0).
CPU time = 0.017 sec                % transformation time

_@ member(X,[a,b,c,d,e]),member(X,[c,d,e,f,g]).        % old constraint

solution = c4(X)              % modular form
c7(d).                        % new clause (c4,c6,and c7 are new predicates)
c7(e).
c6(c).
c6(X0):-c7(X0).
c4(X0):-c6(X0).
CPU time = 0.017 sec

_@ member(A,X),append(X,Y,Z).         % old constraint

solution = c10(A, X, Y, Z)        % modular form
c12(X2, X2, Y3, Y0, Z1):-append(Y3, Y0, Z1).
c12(X0, Y3, [A1|X2], Y3, [A1|Z4]):-c12(X0, A1, X2, Y3, Z4).
c10(A0, [A1|X2], Y3, [A1|Z4]):-c12(A0, A1, X2, Y3, Z4).
                        % recursive clause is defined by fold transformation.
CPU time = 0.017 sec
_%L
 +-- List integrate() traces --+  % history of new predicate
<4,2> c10(A, X, Y, Z) <=> append(X, Y, Z), member(A, X).
<4,2> c13(Y0, Z1, X2, Z4) <=> append(Z4, Y0, Z1), member(X2, Z4).
<5,2> c12(A0, A1, X2, Y3, Z4) <=> append(X2, Y3, Z4), member(A0, [A1|X2]).
<1,2> c4(X) <=> member(X, [c,d,e,f,g]), member(X, [a,b,c,d,e]).
<1,2> c5(X0) <=> member(X0, [c,d,e,f,g]), member(X0, [b,c,d,e]).
<1,2> c8(X0) <=> member(X0, [e]), member(X0, [c,d,e,f,g]).
<1,2> c7(X0) <=> member(X0, [d,e]), member(X0, [c,d,e,f,g]).
<1,2> c6(X0) <=> member(X0, [c,d,e]), member(X0, [c,d,e,f,g]).
<1,1> c2(X0) <=> member(X0, [c]).
<1,1> c1(X0) <=> member(X0, [b,c]).
<1,1> c0(X) <=> member(X, [a,b,c]).

_%p member                  % spy member()
 +++ spy member  +++
_%t                         % normal trace mode

 +++ normal trace on +++
$:-member(X,[b,c,d]). %trace member(X,[b,c,d])
 [0>>member(X_0, [b,c,d])
   <=0-1=member(X, [X|Y]). %apply one program clause
success.
member(b, [b,c,d])
  X = b; %first solution. type ';' to continue
 [0>>member(X_0, [b,c,d])
   <=0-2=member(X, [Y|Z]):-member(X, Z).
 [1>>member(X_26, [c,d])
   <=1-1=member(X, [X|Y]).
success.
member(c, [b,c,d])
  X = c; %second solution. continue
 [1>>member(X_26, [c,d])
   <=1-2=member(X, [Y|Z]):-member(X, Z).
 [2>>member(X_44, [d])
   <=2-1=member(X, [X|Y]).
success.
member(d, [b,c,d])
  X = d; %third solution. continue
 [2>>member(X_44, [d])
   <=2-2=member(X, [Y|Z]):-member(X, Z).
 [3>>member(X_62, [])
   <=3-no= fail member.
no. %no more solution
CPU time = 0.050 sec

$%p >                       % spy constraint transformation
 +++ spy fold/unfold transformation
$%n new                       % change new predicate name into ''new.''
$@ member(A,X),append(X,Y,Z).
```

```
 +++ normal trace on +++
$%n new
$@ member(A,X),append(X,Y,Z). %transfer

1 transfer  member(A_6, X_8), append(X_8, Y_10, Z_12)  : A_6, X_8, Y_10, Z_12
    <1> new predicate new0 is defined.
1=1(new0)  <<- append([], X, X). % unfold
2 transfer  member(A_6, [])  : A_6, X_64
2 => Fail
1=2(new0)  <<- append([A|X], Y, [A|Z]):-append(X, Y, Z). %unfold
2 transfer  append(X_66, Y_68, Z_70), member(A_6, [A_64|X_66])  : A_6, A_64, X_66, Y_68, Z_70
    <2> new predicate new1 is defined.
2=1(new1)  <<- member(X, [X|Y]). %unfold
3 transfer  append(Y_223, Y_68, Z_70)  : Y_68, Z_70, X_221, Y_223
3 => append(Y3, Y0, Z1)
2=2(new1)  <<- member(X, [Y|Z]):-member(X, Z). %unfold
3 transfer  member(X_221, Z_225), append(Z_225, Y_68, Z_70)  : Y_68, Z_70, X_221, Y_223, Z_225
3===fold new0(A, X, Y, Z) <=> append(X, Y, Z), member(A, X) %fold
3 => new0(X2, Z4, Y0, Z1)
2 => new1(A0, A1, X2, Y3, Z4)
1 => new0(A, X, Y, Z) % == member(A,X),append(X,Y,Z)
*** reduce  new1(X2, Y3, Z4, Y0, Z1):-new0(X2, Z4, Y0, Z1). <- new0(A0, [A1|X2], Y3, [A1|Z4]):-new
          =>  new1(A0, Y5, [A1|X2], Y3, [A1|Z4]):-new1(A0, A1, X2, Y3,
Z4).
%reduction (c0 has only one clause)
solution = new0(A, X, Y, Z)
new1(X2, X2, Y3, Y0, Z1):-append(Y3, Y0, Z1).
new1(A0, Y5, [A1|X2], Y3, [A1|Z4]):-new1(A0, A1, X2, Y3, Z4).
CPU time = 0.083 sec
```

10.2 Programming in CAHC

```
_"t2.p?                          % read ''t2.p'' file with echo back
open 't2.p'
member(X,[X|Y]).                 % define member()
member(X,[Y|Z]):-member(X,Z).
append([],X,X).                          % define append()
append([A|X],Y,[A|Z]):-append(X,Y,Z).
inter(A,X,Y):-member(A,X);member(A,Y). % intersection by CAHC

:-inter(X,[p,o,i,p,o,i,p,o,i,p,o,i],[x,c,v,a,s,d,a,s,d,a,s,da]).
Failure.
CPU time = 0.067 sec
:-member(X,[p,o,i,p,o,i,p,o,i,p,o,i]),member(X,[x,c,v,a,s,d,a,s,d,a,s,da]).
Failure.
CPU time = 0.083 sec              % execution by Prolog is the slowest
@member(X,[p,o,i,p,o,i,p,o,i,p,o,i]),member(X,[x,c,v,a,s,d,a,s,d,a,s,da]).
solution = fail.
CPU time = 0.067 sec
```

10.3 JPSG parser

```
 *****  cu - Prolog   Ver. 3.02  *****
 Copyright: Institute for New Generation Computer Technology, Japan 1989
          help ->  %h

_"jpsg.p" % read JPSG parser file
=== open 'jpsg.p'

 ****** end of file *******
CPU time = 0.917 sec

% user input: p() is the top level predicate which draws a parsing
% tree and prints the top category and constraints.

_:-p([ken,ga,naomi,wo,ai,suru]). % ''Ken ga Naomi wo aisuru'' means
 % ''Ken loves Naomi.''

% parser draws a parsing tree
v[syusi]:[love,ken,naomi]---[suff_p]
 |
```

```
  |--v[vs2]:[love,ken,naomi]---[subcat_p]
  |    |
  |    |--p[ga]:ken---[adjacent_p] % Mother (adjacent feature principle)
  |    |    |
  |    |    |--n[n]:ken---[ken] % Right daughter (Head)
  |    |    |
  |    |    |__p[ga, AJA:n[n]]:ken---[ga] % Left daughter (Adjacent)
  |    |
  |    |__v[vs2, SC:{p[ga]}]:[love,ken,naomi]---[subcat_p]
  |         |
  |         |--p[wo]:naomi---[adjacent_p]
  |         |    |
  |         |    |--n[n]:naomi---[naomi]
  |         |    |
  |         |    |__p[wo, AJA:n[n]]:naomi---[wo]
  |         |
  |         |__v[vs2, SC:{p[wo], p[ga]}]:[love,ken,naomi]---[ai]
  |
  |__v[syusi, AJA:v[vs2]]:[love,ken,naomi]---[suru]

cat cat(v, syusi, [], [], [], [love,ken,naomi])
cond % null means no ambiguity
true.
CPU time = 0.217 sec

% ''Ken ga aisuru'' means ''Ken loves
% (someone).'' or ''(someone) whom Ken
% loves.''
_:-p([ken,ga,ai,suru]).

v[Form_865, AJN:Adj, SC:{SubCat}]:SEM_871---[suff_p]
  |
  |--v[vs2, SC:{p[wo]}]:[love,ken,Obj0_571]---[subcat_p]
  |    |
  |    |--p[ga]:ken---[adjacent_p]
  |    |    |
  |    |    |--n[n]:ken---[ken]
  |    |    |
  |    |    |__p[ga, AJA:n[n]]:ken---[ga]
  |    |
  |    |__v[vs2, SC:{p[ga], p[wo]}]:[love,ken,Obj0_571]---[ai]
  |
  |__v[Form_865, AJA:v[vs2], AJN:Adj, SC:{SubCat}]:SEM_871---[suru]

cat cat(v, Form_865, [], Adj_867, SubCat_869, SEM_871)
cond c6(Form_865, SubCat_869, Obj0_571, Adj_867, SEM_871)
true.
% ambiguity is packed in the solutions
% of the constraint c6(...)
CPU time = 0.150 sec

_:-c6(F,SC,_,AJ,SEM). % solve c6(...)
  F = syusi  SC = [cat(p, wo, [], [], [], Obj00_34)]  AJ = []  SEM =
[love,ken,Obj00_34];
% first solution corresponds to ''Ken
% loves (someone).''
  F = rentai  SC = []  AJ = [cat(n, n, [], [], [], inst(Obj0_34,
Type1_36))]  SEM = inst(Obj0_34, [and,Type1_36,[love,ken,Obj0_34]]);
% second solution corresponds to
% ''(someone) whom Ken loves.''
no.
CPU time = 0.017 sec
```

References

[1] T. GUNJI. *Japanese Phrase Structure Grammar.* Reidel, Dordrecht, 1986.

[2] S. M. Shieber. *An Introduction to Unification-Based Approach to Grammar.* CSLI Lecture Notes Series No.4. Stanford:CSLI, 1986.

[3] H. SIRAI and K. HASIDA. Jyokentsuki Tan'itsu-ka (Conditioned Unification). *Computer Software*, 3(4):28–38, 1986. (in Japanese).

[4] H. TSUDA, K. HASIDA, and H. SIRAI. cu-Prolog and its application to a JPSG parser. In *Proc. of Logic Programming Conference*, pages 155–164, Tokyo, 1989.

[5] H. TSUDA, K. HASIDA, and H. SIRAI. JPSG Parser on Constraint Logic Programming. In *Proc. of 4th ACL European Chapter*, pages 95–102, 1989.

◎cu-Prolog V2ソースファイル概要

cu-Prolog

1990.9.1

ICOT第三研究室　　津田 宏

§1. ヘッダーファイル(*.h)

cu-prologのヘッダファイルは以下の5つである。

・include.h

データ構造体の定義、各種マクロの定義、ヒープ及びスタックサイズの設定を行う。全ソースファイルから呼ばれている。

・funclist.h

*.cファイルに含まれる関数の一覧。これを読み込むことにより、どのモジュールに入っている関数も自由に使うことができる。include.hから呼ばれている。

・varset.h

全モジュールで共通して使われるグローバル変数の定義を行う。main.cで呼ばれる。

・globalv.h

グローバル変数の外部参照用のヘッダ。main.c以外のモジュールで使用する。

・sysp.h

cu-Prologのシステム組み込み述語を表す変数の外部参照用ヘッダ。変数の定義自体はsyspred.cで行っている。syspred.c以外のモジュールで、組み込み述語に対する操作が必要なものに使用する。

§2. Cソースファイル(*.c)

現在、cu-PrologのCソースファイルは以下のモジュールから成り立っている。

・main.c

prologのトップレベル、エラーの処理を行う。

・mainsub.c

main.cのサブプログラム集。ヘルプ、オプション(%)コマンド、ファイル操作等

・new.c

ユーザーヒープ、システムヒープ、ユーザースタックの定義。ヒープ、スタック関係の関数がすべて定義されている。ヒープ上に構造体をallocする関数は N*** という名前で作られている。

・read.c

ホーン節の読み込みについての関数が定義されている。他モジュールとのやりとりは主に cbuf(1文字)、nbuf(文字列)の2つのグローバル変数に基

づいて行う。

・print.c
節、項などの表示関係の関数が定義されている。ほとんどの関数は、項や節を表す変数と、その環境(主にeという変数で表す)とをペアにして呼ばれる。

・refute.c
いわゆるprologの反駁本体の処理を行う。

・unify.c
通常のユニフィケーションの処理を行う。

・defsysp.c, syspred1.c, syspred2.c, jpsgsub.c
cu-Prologのprologの組み込み述語の処理を行う。
defsysp.cでは組込み述語のエントリを定義
syspred1.cではassertなど一般的な組込み述語の処理
syspred2.cでは数値、文字列演算の組込み述語の処理
jpsgsub.cではjpsgパーザ用の組込み述語の処理

・cunify.c
制約変換に必要なツール関数集。up**()関数群は、ユーザーヒープに格納されている一時的な素式や節を、システムヒープにコピーするのに使う。greater()は2つの素式の順序を計算する。integrateのトレースを格納するときにこれを用いて節をソートすることで能率を上げる。goodterm()は基礎項からなる素式をprologとして実行し、true/failを返す。

・genfunc.c
制約変換時に新たに作られる節の名前を作り出す。通常はc0,c1...という名前のものができるが、genname□というグローバル変数を書き換えることにより変えることができる。

・modular.c
制約変換の呼び出し関数、startmodular()をはじめ、modularizeの処理を行う。

・split.c
制約変換処理で、素式列を変数の同値類により分割する処理を行う。現在は与えられた列を静的に分割するだけだが、将来は増進的に分割が行えるように改良して行きたい。

・reduce.c
制約変換でできた節を簡約化する処理を行う。定義節がただ一つしかないの述語を本体に含む節を強制的に書き換えている。

・integ.c
制約変換の主要部、integrateの処理を行う。integrate()という関数がそれ。少し長すぎるので、分割したい。

§3. コンパイルの方法

cu-Prologの実行コードを得るには、以上のモジュールを分割コンパイルしリンクで合わせるだけである。ただし、使用するOS、コンパイラによりinclude.hの最初の、

```
#define MSDOS ????
```

の値を書き換える必要がある。

MS-DOSのCコンパイラでスモールモデルの場合。
(intが16ビット、ポインタサイズが16ビット)
例:MS-C ver.4、Turbo-C ver.1.5

```
#define MSDOS   1
```

MS-DOSのCコンパイラでラージ(ヒュージ)モデルの場合。
例:MS-C ver.4(large) Turbo-C ver.1.5(huge)
(intが16ビット、ポインタサイズが32ビット)

```
#define MSDOS   2
```

UNIXのCコンパイラの場合。
例:Ultlix
(int、ポインタサイズ共に32ビット)

```
#define MSDOS   0
```

デフォルトはUNNIXなので,単にmakeすればよい.

なお、実行していて
user heap overflow　がでるなら HEAP_SIZE
user stack overflow がでるなら USTACK_SIZE
system heap overflowがでるなら SHEAP_SIZEの値をそれぞれ大きくするとよい。

また、include.h のCPUTIMEの値もコンパイラにより書き変える必要がある。
UNIX4.2/4.3 BSDでtimes()関数があり,1/n単位でCPU-timeを計測する場合には,

```
#define CPUTIME n     (普通は60)
```

とする。
それ以外には 0 とする。この時は,実行時のcpuタイムが表示されない。

また、SUN4のUNIXの場合は、

```
#define CPUTIME 1000000
```

と指定する。

2. include.h ドキュメンテーション

cu-Prolog

第1版 88.7.2
第2版 88.11.29 (ver.2.00)
第3版 90.3.26 (ver 3.0) 90.6.22(ver3.01) 90.7.14(ver.3.02)

§1.概要

このヘッダファイルでは、全ての構造体が定義され、重要なマクロや変数も定義される。全てのモジュールにも取り込まれる。

§2.#define 文 説明

MSDOS

使用する機種、コンパイラを設定する。変数領域が大きすぎるなどのコンパイルエラーが起こったら、この値が正しいかどうかまず調べることが必要である。

1 : MS-DOSスモールモデル(int、ポインタサイズ共に16ビット)
2 : MS-DOSラージ(ヒュージ)モデル(int16ビット、ポインタサイズ32ビット)
0 : Unix(int、ポインタサイズ共に32ビット)

の値に設定する。

CPUTIME

UNIX4.2/3 BSDのCコンパイラでサポートされている、times()関数(各プロセスのcpuタイムをlong整数で返す)がない場合は0にする。times()関数が1/n秒単位で計測する時は(普通は1/60単位),

#define CPUTIME n

と設定する.この値を指定した場合,トップレベルで実行時間を表示する。
SUN-4の場合には

#define CPUTIME 1000000

とする。この場合にはCPUタイムの計測にclock()関数を使う。

HEAP_SIZE

ユーザーヒープの大きさを設定する。ユーザーヒープにはprolog反駁実行時のノード等、一時的データ構造が格納される。user heap overflowのエラーが起こったらこの値を大きくする。

SHEAP_SIZE

システムヒープの大きさを設定する。システムヒープには、プログラムホーン節、制約単一化により新たに生成された述語の定義、integrateの履歴、などが格納される。制約単一化機構を使うためには、この大きさが十分なければならない。system heap overflowのエラーが起こったらこの値を大きくする。

USTACK_SIZE

ユーザースタックの大きさを設定する。ユーザースタックは、主にprologまたは制約単一化で、ポインタの付け替えの際に用いる。utack overflowエラーが起きたらこの値を大きくする。prologでかなり深い推論をし

ない限りはこのエラーは滅多に起こらない。

NAME_SIZE
　　ストリング領域のサイズ。

ENV_SIZE
　　pair構造体のサイズ．MS-DOS huge model の時のみ4．他は2.

tputc()
tprint?(X,V1,V2……)
　　ログファイルへの出力も兼ねたprint関数マクロ。ソースファイルのprint文にはprintfを使わずにすべてtprintを使う。Xにはフォーマット、V?は引数を入れて用いる。引数の数により異なったマクロを使わなければならないことに注意。lfpはログファイルポインタを表す変数で、通常はNULL、ログをとるときはファイルへのポインタが入る。

readword(S)
　　ログファイルへの出力を兼ねた、入力用マクロ。Sには文字列(配列)を表す変数をとる。

#define skipline : CRが来るまで行を読み飛ばす。

NL : 改行。ログファイルにも対応している。

#define NUMSTRING-TYPE
　　数値、文字列の項のタイプ
#define FILE_TYPE
　　ファイルを表す項のタイプ

alpha : 文字バッファcbufがアルファベット、空白文字であるかを検査する。if文の条件に使用。
quotesign : cbufが引用符(',")であるかチェックする。
white : cbufが空白文字であるかチェックする。

advance : 空白文字列を読み飛ばし、次の文字をcbufに入れる。

KEYIN
　　キーボード入力状態かチェックする。fp(入力ファイルポインタ)がstdinか調べている。if文の条件に使用する。

VT-100 Escape Sequence
　　画面表示のためのエスケープシーケンスである

・var構造体のタイプに関するマクロ
以下の5つのマクロはvar構造体に関するものである。varは変数型を表すデータ構造であるが、プログラム中ではキャスト演算子によりterm型として使われることが多い。ファンクタ、述語の引数は、項、変数どちらもとるためである。従って、プログラムの変数が変数型であるかどうかを調べるには、以下のようなマクロを使う必要がある。tはいずれもterm型の変数を表す。

VAR_TYPE 定数。vが変数の時、v->v_type==VAR_TYPE
isvar(t) : tが変数であるかどうかのチェック
vnumber(t) : tの変数番号(節の中で何番目に現れた変数か)を得る
vname(t) : tの変数名を得る
voccurence(t) : tが節の本体にいくつ現れているかを得る
vlink(t) : tの(v_linkでつながっている)次の変数をterm型として返す。
vconsraint(t) : tに関する制約を返す。

・term構造体のタイプに関するマクロ
NUMSTRING_TYPE :項tが数値又は文字列の時t->type.ident==NUMSTRING_TYPE
FILE_TYPE:項tがファイルを表す場合、t->type.ident==FILE_TYPE

is_atomic(t) : tが数値又は文字列であるかのチェック
is_constant(t) : tが定数アトムであるかのチェック
is_num(t) : tが数値であるかのチェック
is_string(t) : tが文字列のチェック
is_int(t) : tが整数かのチェック
num_value(t) : tが数値を表す時、その値
str_value(t) : tが文字列を表す時、その文字列へのポインタ
is_file(Term): Termがファイルを表すかのチェック
is_readable(FP): ファイルポインタFPがreadableかチェック
is_writable(FP): ファイルポインタFPがwritableかチェック
isconst(t) : tが変数を含まない項(定数、定数複合項)であるかのチェック
notconst(t) : tが変数を含む項であるかのチェック
isatom(t) : tが定数(アトム、文字列、数値)であるかのチェック
Arg(T,N) : 項Tの(N+1)番目の引数項
Arg1(T) : 項Tの1番目の引数項
Predname(T) : 項Tの述語名

・変数の束縛をたぐる手続き
down(p,t,e)
struct pair *p,*e
struct term *t
　　pは何も代入していない変数、(t,e)で調べたい項を指定して呼ぶ。(t,e)の束縛をたぐり、free変数に束縛している場合には、(t,p)によりその変数を表す。(t,e)がfree変数以外(複合項、定数)に束縛しているときは、pはNULLで(t,e)は束縛先を示す。基本的な使い方としては、ある項(t,e)がfree変数かどうかを検査するときに用いる。down(p,t,e)を行い、pがNULLの時はfree変数。
downを使う際の注意として、
　　down(e[i]、t->t_arg[k],p[j])
のように、変数以外を引数に指定してはならないことである。この場合は、
　　e1=e[i]
　　t1=t->t_arg[k]
　　p1=p[j]
　　down(e1,t1,p1)
とする。downの実行後、e1,t1,p1いずれもが書き換えられる可能性がある。

・文字列領域への格納を指定するフラグ

TEMPORAL
ETERNAL
STINGY

・文字列のタイプ
QUOTE_NAME : 'で始まり'で終わる任意の文字列
STRING : "で始まり"で終わる任意の文字列
NUMBER : 数値(先頭には+または-がつくことがあり、小数点も一つまで含む)
VARNAME : 変数を表す(先頭が大文字アルファベットではじまる)
NAME : 上記以外の文字列のタイプ

・述語のタイプ
USERFUN : ユーザ定義述語、制約変換でできた新述語
SYSFUN : システム組み込み述語
SPYFUN : スパイされている述語
REDUCEDFUN : 制約変換の簡約により消えた述語
FINITEFUN : 再帰的でない述語
TEMPFUN : 一時的な述語
NONFUNC : (組込み)述語で関数的でない(バックトラックで複数解を返す)もの
SYSFUN2 : SYSFUN + NONFUNC

・述語構造体のタイプに関するマクロ
systemfun(F) : func型Fを組み込み述語とする。
userfun(F) : func型Fをユーザ定義述語とする。
issystem(F) : Fが組み込み述語かのチェック
isuser(F) : Fが組み込み述語でない(ユーザー述語である)かのチェック
isfunc(F) : Fが関数的(バックトラックしない)述語であるか
isnonfunc(F) : Fが関数的でない(バックトラックする)述語であるか
is_funcsys(F) : Fが関数的組込み述語であるか
is_nofuncsys(F) : Fが関数的でない組込み述語であるか
spyfun(F) : Fのスパイフラグをonにする。
nospyfun(F) : Fのスパイフラグをoffにする。
isspy(F) : Fがスパイされているかどうか?
isnospy(F) : Fがスパイされていないかどうか?
spychange(F) : Fのスパイフラグを変える(on<->offのスイッチである)
reducedfun(F) : Fのreduceフラグをonにする。
isreduced(F) : Fがreduceされているかどうか?
isnoreduced(F) : Fがreduceされていないかどうか?
finitefun(F) : Fは有限の述語である。
isfinite(F) : Fが有限述語かどうか?
isinfinite(F) : Fが有限でない述語(再帰を含む述語)かどうか?
newpred(F) : Fは制約変換で作られた新述語である
isnewpred(F) : Fが新述語であるかどうか?
isnotnewpred(F) : Fが旧述語であるかどうか?
isallunit(F) : Fの定義が全てユニット節からできているかどうか。

・set構造体に関するマクロ
is_unitclause(s) sがユニット節(本体が空)であるかどうか

・トレース表示のマクロ

Notrace_mode トレースモードをやめる
Normaltrace_mode ノーマルトレースモードにする
Steptrace_mode ステップトレースモードにする
Is_Notrace トレースモードでないか
Is_Normaltrace ノーマルトレースモードか
Is_Steptrace ステップトレースモードか
Is_Trace (どちらかの)トレースモードか
Msolvable_mode M-solvableモードにする
Modular_mode 全モジュラーモードにする
Is_Modular モジュラーモードか
Noreduce_mode 簡約しないモードにする
Reduce_mode 簡約モードにする
Is_Noreduce 簡約モードでないか
Is_Reduce 簡約モードか

TB : トレース時に次のTEまでが実行される。
STB(F) : Fがスパイ述語の時、次のSTEまでがトレース時に実行される。
TE : トレースエンド。TBではじめたルーチンの最後におく。
STE : トレースエンド。STBではじめたルーチンの最後におく。

TTRANSB : 制約変換のトレース表示begin
TTRANSE : 制約変換のトレース表示end

・構造体のアロケーションに関するマクロ
snew(s)
sには構造体名をとる。sの示す構造体をsheap(システムヒープ)上にallocする。

new(s)
sの示す構造体をheap(ユーザーヒープ)上にallocする。

MFAIL : 制約変換が失敗した時に返る値。

TRUE = 1
FAIL = 0

・syspred.cで組込み述語を実行ルーチンの返値
SYSTRUE : 組み込み述語が成功した
SYSFAIL : 組み込み述語が失敗した

・refute.cでノードの探索の状態を表すフラグ
DOWN 下向き
UP 上向き
BACKTRACK バックトラック中

HASH_SIZE : ハッシュテーブルのサイズ

NAME_MAX : 文字列バッファの大きさ(定数名、述語名の最大長)

・ヒープ、スタック
shp[SHEAP_SIZE] : システムヒープ(プログラム節が入る)

shp　　：システムヒープポインタ
heap[HEAP_SIZE]：ユーザヒープ(refutationのノード等一時的な構造が入る)
hp　　：ユーザヒープポインタ
ustack[USTACK_SIZE]：ユーザスタック
usp　　：ユーザスタックポインタ
nheap[NAME_SIZE]：文字列ヒープ
nhp　　：文字列ヒープポインタ
nheap_t[NAME_SIZE]：一時的文字列ヒープ
nhp_t　：nheap_tへのポインタ
hash_list[HASH_SIZE]：ハッシュテーブル
※名前がfnameの述語はhash_list[hash(fname)]に格納されている。各hash_list[i]は、逆abc順、同一名の述語は引数の小さい順に並べている。

§3.構造体定義 説明
凡例: S :システムヒープのみに格納される。
　　H :ユーザーヒープのみに格納される。
　　無印:両ヒープに格納される。

・変数型 S
```
struct var {
        int v_type;      変数の場合は1、それ以外は0
        int v_number;  各節ごとに変数番号(0,1,2...)がつく。
        int v_head_occur; 節の頭部に出現する回数
        int v_occurence; 節の本体に出現する回数
        char v_name[16]; 変数名
        struct var *v_link;      次の変数へのリンク。最後はNULL。
        struct clause *v_constraint;  CAHCの制約
};
```

※ v_head_occur, v_occurenceは第3版でつけ加わったラベル。制約変換のヒューリスティックスに用いる。

・関数(ファンクタ、述語)型 S
```
struct func {
        int f_arity,    引数の数、0の場合は定数。
          f_number,     関数番号(項の大小関係に使う)
          f_mark;       述語の状態(システム、スパイ等)
        int f_setcount; 定義節の数
        int f_unitcount; 定義節のうちユニット節(本体が空)の数
        char f_name[16];        関数(述語)名
        union{
          struct set *f_set;    述語の定義へのポインタ
          SYSFUNC f_sysfunc;    組み込み述語実行ルーチンへのポインタ
        }def;
        struct func *f_link;  次の関数へのポインタ
        struct itrace *f_integ; integrateトレースへのポインタ
        int f_cbind[1]; f_cbind[i]は(i+1)番目の引数が定義のヘッドのうち何回定数項にバインドするかを示す。
};
```

※ f_setcount, f_unitcount, f_cbind口は第3版になってつけ加わった構造体ラベルである。制約変換のヒューリスティックスに用いる。

・項型

```
struct term {
        union{
          struct func *t_func;  述語(ファンクタ)へのポインタ
          int ident;     atomic(数値、文字列)な項では=0
        }type;
        int t_arity;              引数の数(※1)
        union{
          struct term *t_arg[1];          引数項へのポインタ。
          float n_value;          数値を表す時その値
          char *s_value;          文字列を表す時、それへのポインタ
        }
};
```

(※1) t_arityは、> 0のとき: 引数の個数。 定数複合項ではない
= 0のとき: 定数(アトム)
< 0のとき: 引数の個数の負符号。定数複合項

※※項の分類

1. 定数アトム: a,abc のように小文字から始まる文字列。大文字から始めたり、途中にスペースを入れたい時は、'A pen is'のように'で囲む。
tが定数アトムの時、t->t_arity == 0 である(マクロis_constant(t))。
t->type.t_func->f_name により定数名を得る。
定数アトムは引数の数0の述語とする。

2. リスト: リストファンクタは大域変数LISTで表される。リストの末尾(□)はNILで示される。

3. アトミックな項 (is_atomic(t)でチェック可)
3-1. 数値
t->type.ident == NUMSTRING_TYPE、t->t_arity == 1(float)
t->t_arity = 0 (int)の場合である(マクロis_num(t)でチェックできる)。
num_value(t)により数値の値を得る。

3-2. 文字列 : ""で囲まれた任意の文字列
t->type.ident == NUMSTRING_TYPE, t->t_arity == 2の場合である
(is_string(t)マクロでチェック可)。
str_value(t)で文字列の中身を得る。

4. 定数項: 定数複合項とは変数を含まない項のことである。
上の1,3,4および、定数を含まない複合項より成り、isconst(t)でチェックできる。
アリティkの複合項が変数を含まない時
t->t_arity = -k
である。

・節型

```
struct clause {
        struct term *c_form;  項へのポインタ
        struct clause *c_link; 次の節へのポインタ
};
```

・定義型(ホーン節の定義をまとめる) S

```
struct set {
        struct clause *s_clause; 定義ホーン節本体
        int s_vnumber;           定義に含まれる変数の個数
        int s_bodynumber;        定義節の本体のリテラルの数
        struct term *s_vlist;    定義に含まれる変数リストへのポインタ
        strcut clause *s_constraint;    CAHCの制約部分
        struct set *s_link;      次の定義へのポインタ
};
```

・環境型(ストラクチャ・シェアリング用の、変数に対する環境) H

```
struct pair {
        struct term *p_body;    対応する項
        struct pair *p_env;     対応する環境
};
```

・スタック型

```
struct ustack {
        int *u_addr;    アドレス
        int u_val;      値
};
```

・ノード型 H (導出木のノード。refute.cで主に使う)

```
struct node {
        struct clause *n_clause; ゴールリテラル
        struct pair *n_env;      ゴールの変数環境
        struct set *n_set;       定義ホーン節へのポインタ
        struct node *n_link,     親ノード
                    *n_last;     バックトラック先のノード
        struct constraint *n_constraifnt;       制約
        int *n_hp,               ノード定義時のヒープポインタ
          n_count,               ノード番号(トレース時に使う)
          n_spy;          ノードがスパイ時1、そうでない時0
        struct ustack *n_usp;    ノード定義時のスタックポインタ
};
```

・制約型 H (CAHCの制約変換処理で使う)

```
struct constraint{
        struct clase *co_clause;        制約(リテラルの並び) S
        struct pair *co_env;            制約と変数を結ぶ環境
        struct term *co_var;            制約に含まれる変数のリスト
```

```
        int     co_vnumber;             制約に含まれる変数の個数
}
```

・環境付き節型 H(cuで使う)

```
struct eclause {        /*  environment + clause(copy) */
        struct term     *c_form; 項へのポインタ
        struct pair     *c_env;  項の環境
        struct eclause  *c_link; 次の節へのポインタ
};
```

・全変数型 H (cuでcnode毎に変数をかき集める)

```
struct allvar{          /* var structure for CU */
        struct term     *v_var,  古い変数へのポインタ
                        *v_svar;  新しい変数へのポインタ
        struct pair     *v_env;   環境
        struct eclause  *v_eclause; この変数を含む節(split.c)
        struct allvar   *v_link; 次の変数へのポインタ
        int             v_flag;   分割用ワークフラグ
};
```

・制約変換用ノード型 H (integrateとmodularizeの変数の受渡しにつかう)

```
struct cnode {
        struct eclause  *n_eclause; 節
        struct pair     *n_env;   対応する環境
        struct set      *n_set;   取り上げている定義
        struct cnode    *n_link;  親ノード(分割時)
        int             n_vnumber; 変数の数
        struct allvar   *n_avlist; 全変数リストへのポインタ
};
```

・integrateトレース型 S

```
struct itrace{          /* integrate trace */
        struct clause   *it_clause; 定義節
        int             it_vnumber,  含まれる変数の個数
                        it_cnumber;  含まれる項の個数
        struct itrace   *it_link;    次のトレースへのポインタ(%L)
};
```

※ it_vnumber, it_cnumberはfold変換のサーチのキーになる。

3. varset.h globalv.h ドキュメンテーション
cu-Prolog

第1版 88.7.2
第2版 88.11.29 (ver.2.00)
90.3.14 (ver3.0)

1.概要

複数のモジュールで共通に使うグローバル変数は、1箇所で管理しておいた方が見通しがよくなる。varset.hはグローバル変数を初期定義するヘッダでmain.cで読み込まれている。globalv.hはmai n.c以外のモジュールで、外部変数としてグローバル変数を参照する際のヘッダである。

2.変数名の説明

fp

入力ファイルポインタ。キーボード入力時は、stdin外部ファイルからの入力時は、そのファイルへのポインタ、が入る

wfp

出力ファイルポインタ。
通常出力時は、stdout
無出力時(外部プログラムの読み込みの時)は、NULL
%wで定義節を外部ファイルに書き出すときは、ファイルへのポインタ
が入る。

lfp

ログファイルポインタ。通常時は、NULLログをとるときは、ログファイルへのポインタ、が入る

tty

テレタイプかどうか?

cbuf

キャラクタバッファ。入力された1文字をしまう。

nbuf

文字列バッファ。入力文字列(16文字)をしまう。

tflug

トレースフラグ。
0:トレースなし
1:ノーマルトレース(停止なし)
2:ステップトレース(ゴール毎に停止する)

sflug

モジュラー変換モードフラグ
0:M-Solvableモード
1:全モジュラーモード

cutf
　　カットフラグ。refuteでカットを読み込むと1になる。
　　cf. Mnode() (in new.c)

redflag
　　制約変換の簡約化フラグ 0でoff, 1でon

refute_node_count
　　制約変換ルーチンのトレース表示切替え(TTRANSB,TTRANSE)に用いる。
0モードの場合は-1、Prolog中ではノード順位になる。

v_number
　　変数の数。Nvar()関数のなかで1ずつ増える。

v_list
　　変数リストのトップ。Nvar()関数で変わる。

f_list
　　述語リストのトップ。const_list定数(arity=0の述語)リストのトップ。

n_last
　　prologで、バックトラックで戻るノードを指定するときに使用。
Mnode()で書き換えられる。

newf_list
　　integrateのトレースのリストのトップ。

Def_Modified
　　ユーザ述語定義が変化した(新定義、assert/retract後)ことを示すフラグ。check_recursion()で用いる。

Refcount
　　refuteのカウンタ、これ以上深いレベルの探索は強制的にfail扱いにする。%cコマンドで書き換えることができる。

LIST
　　リストファンクタ

CUNIFY
　　prologの組み込み述語としての制約単一化ファンクタ

CUT
　　cut(!)項

NIL
　　リストの最後、つまり□。

FAIL

fail項

genname□
制約単一化でできる変数の名前。genfunc.c参照のこと。

logfile□
ログファイル名をしまう。

☆ main.cドキュメンテーション
cu-Prolog

第1版 88.7.3、11.8
第2版 88.11.29 (ver.2.00)
89.6.28
90.3.14 (ver3.0) 90.7.13(ver3.02)

§1.概要

このモジュールはcu-Prologのトップレベルの処理を行う。

§2.変数、マクロ説明

reset

jmp_buf変数。エラーからトップレベルに大域脱出(longjmp)をするときに使う。

h,sh,u,shp_save,shp_init

それぞれ、ユーザーヒープ、システムヒープ、ユーザースタック、定数リストの値をセーブするのに用いる。

§3.関数の説明

main()

cu-prologのトップレベルの処理を行う。%tコマンドを追加するときは、case文の並びに付け加えればよい。ラベルTOPLEVEL以下がトップレベルのループ。入力カーソルの形状はトレースモードにより異なる。通常は _ノーマルトレース時(tflag=1) は $ステップトレース時(tflag=2) は >である。

error(s)

sにはエラーメッセージが入る。トップレベルの処理に大域脱出する。

prepare()

グローバル変数、組み込み述語などを初期化する。UNIXの場合はsignal(SIGINT,SIG_IGN)で^C割り込みをカットしている。

open_title()

cu-Prologのオープニングタイトル表示

init_status()

システムの状態を立ち上げ時に戻す。

push_status()

%&コマンド(現状態記憶)処理を行う。主な大域変数をセーブする。pop_status()によりシステムをこの状態に戻すことができる。

pop_status()

%$コマンド(元の状態復帰)処理を行う。push_status()で設定した状

態にシステムを戻す。

print_constant()
　　デバッグ用隠しコマンド。%Xにより定数を表示する。

systemcommand(c)
　　%コマンドを実行する。cには%の次の1文字が入る。

garbagecollect()
　　静的なガーベジコレクションを行う。push_status()以降に作られた、ユーザー述語をTEMP.###という一時ファイルに書き出し、pop_status()で状態復帰した後、TEMP.###を再度読み込む。

edit_predicate()
　　デバッグ用隠しコマンド。%Yによりエディターでプログラム節をエディットできる。

trans_routine()
　　モジュラー変換ルーチン、@c1,c2,...,cn. の処理を行う。

questionclause()
　　質問節、:-g1,...,gm,c1,c2,...,cn. の処理を行う。実行中にできた新述語はf_listからのリストで格納されている。
初期ノードを作成した後、refute()を呼ぶ。
実行後、index_funclist()によってハッシュテーブルに登録する。

defclause()
　　α :-β1,β2,...,βm;C1,C2,...,Cn. の形の節(定義節)の読み込みと、システムヒープへの定義のセットを行う。

defnewfunc()
　　fold変換で用いる新述語定義文。H=B1,B2...Bn.を読み込む。未サポート。

☆ mainsub.c ドキュメンテーション
cu-Prolog

第1版　88.7.3, 11.8, 11.10
第2版　88.11.29(ver.2.00) 89.6.28(ver.2.2)
89.11.7 (ver.2.4)
90.3.14 (ver.3.0)

§1.概要

このモジュールはPrologのトップレベル(main.c)から呼ばれるサブプログラムから成っている。

§2.関数の説明

putcursor()
prologのトップレベルのカーソルを表示する。
通常時は、tflag=0で、 _
ノーマルトレース時は、tflag=1で、 $
ステップトレース時は、tflag=2で、 >である。

stepswitch()
%sコマンドによる、ステップトレースon/offのスイッチを行う。
tflagは、
0:トレースなし
1:通常トレース
2:ステップトレース
を表している。

traceswitch()
%tコマンドによる、ノーマルトレースon/offのスイッチを行う。

reduceswitch()
%rコマンドによる、モジュラー変換の簡約化モードon/offのスイッチを行う。オンの場合は、制約変換できた新述語のうち、定義節が一つしかないものは簡約化される。

・int decode_pname(fname)
トレースや述語表示の際の、ユーザ入力 <述語名>/<アリティ> をfnameに取り、fnameを述語名のみにし(/を'\0'と置き換える)、アリティを返す。
アリティ省略時は-1を返す。

spyswitch(fname)
%pコマンドに対応。fnameで表される述語のスパイフラグをon/offする。spychange()はinclude.h参照のこと。
fname="*" のときは、全述語スパイオン.
fname="." のときは、全述語スパイオフ
fname="?" のときは、スパイ述語の表示
それ以外の時は、fname/arityという述語名の述語のスパイをオン・オフする。

allspy(n)

%t,%uコマンドで用いる。n=1のときは、全述語のスパイフラグをonする。nが他の値の時には全述語のスパイフラグをoffにする。

#define is_const_functor(f) ((f->f_arity == 0) && (f->def.f_set == NULL))

0引数述語(関数)が、定数を表しているかチェックする。

showdef(fname)

%d(display)コマンドに対応。fnameには%dに続いて入力された文字列が入って呼ばれる。fnameの値により4通りがある。

(1)fname = "/"

reduceされた述語も含めて、全ての定義を表示する。

(2)fname = "."

reduceされた述語を除いた述語の全定義を表示する。

(3)fname = "?"

述語の名前、各種フラグ状況を表示する。

組み込み述語には *
スパイされている場合は +,
reduceされている述語には -
再帰を含む述語には /

のマークがそれぞれつく。

(4)fnameが上のいずれでもないとき

fname/arityで示される述語の定義のみを表示する。その名前の述語が存在しない場合は、その旨表示する。arity省略時はfnameを述語名に持つ全ての述語の定義を表示。

loghandle(fname)

%lコマンドに対応している。fnameには引数の文字列が入っている。fnameが"no"の場合は、ログファイルをcloseする。それ以外の場合はfnameで示される名前のログファイルをオープンする。同名のファイルが存在する場合にはオープンできない等のエラーチェックも行う。

helpmenu()

ヘルプメッセージを出力する。

freeheap()

システムヒープの残りを表示する。

init_syspred()

システム組み込み述語を初期化する。main.cのprepare()から呼ばれる。

filewrite(n)

%wコマンドに対応。nにはファイル名が入って呼ばれる。簡単なエラーチェックを行った後、wfpを出力ファイルに変えて、ユーザー述語のうちreduceされていないものの全定義を表示するだけである。

disp_func_def(f_from,f_to)
f_list上で、f_fromからf_toまでのユーザー定義で、簡約されていない述語の定義を表示する。

set_inputfile(n)
nにはファイル名が入る。読み込みファイルポインタfpをnに設定する。

readfile()
"file name"コマンド(ファイル読み込み)処理を行う。

set_eof()
ファイル読み込みのEOF処理を行う。

・check_recursion()
全ユーザ定義述語の再帰関係をチェックする。大域変数Def_Modifiedが1の場合にのみ再チェックを行うようにしている。Def_Modifiedは、add_set()および、assert/retract述語の実行により1になる。
check_recursion()を呼ぶのは、refuteの前、@コマンドによる制約変換の前、%dコマンドの前、である。再帰関係のチェック法は、
(1)全てのユーザ述語の内、全ての定義がユニット節からなるもののみfiniteフラグを立てる。組込み述語についてはタイプ1(関数的)のもののみfiniteとなっている。
(2)全定義節の本体が空または全てfiniteフラグの立った述語からなる述語のfiniteフラグを立てる。
(3) (2)を繰り返し、フラグの変化がなくなるまで続ける。

・check_unitpred(f)
述語fがすべてユニット節で定義されている時にfのfiniteフラグを立てる。

・check_all_unif(fl)
述語リストflのそれぞれの述語につきcheck_unitpred()を実行する。

・rec_to_finite()
finiteでない述語の定義節の本体が全てfinite述語からなっている時、述語のfiniteフラグを立てる。一つでもフラグが立った場合、グローバル変数REC_to_FINITEの値は1になる。

・int is_body_finite(f)
述語fの全ての定義節の本体が全てfinite述語から成っている場合はTRUE、そうでないときはFALSEを返す。

ちなみに有限な述語の定義は次である。
定義節が、ヘッドのみからなる述語は有限
定義節のボディが有限な述語から成る述語は有限

oscommand()
OSのコマンドインタープリンターを起動する。

delete_tmp()

一時ファイルを消去する。

quit_prolog()

cu-Prologを終了する。

以下の2関数は、計算時間表示ルーチンである。times()がサポートされている
Cコンパイラ(UNIX 4.2/3 BSD)でないと動かない。include.hの

```
#define CPUTIME ???
```

文で、コンパイラの種類をセットする。使い方は、時刻を計りたいルーチンの
前で、settimer()し、計算の直後にprinttime()する。SUN-4の場合は

```
#define CPUTIME 1000000
```

と指定する。

printtime()

タイマーの値を表示する。

settime()

タイマーをゼロにセットする。

☆ new.c ドキュメンテーション
cu-Prolog

第1版　88.7.3
第2版　89.6.28
89.11.7 (for Ver.2.4)
90.3.13 (ver.3.0)

§1. 概要

　　このモジュールにはヒープ、スタック等のメモリー管理をおこなう関数が入っている。

§2. 変数の説明

TERM_SIZE, FUNC_SIZE, POINTER_SIZE, PAIR_SIZE :
　　各構造体の(intの大きさで計った)サイズ

Termalloc(a)　アリティaの項をシステムヒープ上にallocする。
tempterm(a)　アリティaの項をユーザヒープ上にallocする。
funcalloc(a)　アリティaの関数をシステムヒープ上にallocする。

sheap□
　　システムヒープ。プログラムホーン節、制約変換時に作られた新述語の定義、履歴等が入る。ガーベジコレクションは行っていないのでかなり大きいサイズが必要である。

*shp
　　システムヒープポインタ。

heap□
　　ユーザーヒープ。prologの反駁時に一時的に作られるデータ構造(例:ノード)などが入る。

*hp
　　ユーザーヒープポインタ

ustack□
　　ユーザースタック。ポインタを付け換えたりするとき、古いポインタをスタックに積んでおく。upush(), undo()により操作。

*usp
　　ユーザースタックポインタ

FNUMBER
　　関数番号。初期値は0で、新述語定義毎に1ずつ増える。制約実式列をソートする時に使う。

§3.関数説明
・hash(fname)

述語名fnameに対応するハッシュ関数の値を返す。hash_list[i]からはじまるリストにその関数が含まれている。

・salloc(n)
nワード(int型のビット数)システムヒープ上にとる。SHEAPBOTTOMはシステムヒープ領域の最初のアドレス、SHEAPTOPは最後のアドレスである。

・alloc(n)
nワード、ユーザーヒープ上にとる。HEAPBOTTOMはユーザヒープ領域の最初のアドレス、SHEAPTOPは最後のアドレスである。

・nalloc(n,flag)
文字列へのポインタn、flag(={STINGY,ETERNAL,TEMPORAL})をとる。
flag==STINGYのときはnを返す。
文字列nの述語fがハッシュテーブルに存在する時は、f->f_nameを返す。
flag==ETERNALの時は、nheap□上に文字列をコピーして、それへのポインタを返す。
flag==TEMPORALの時は、nhepa_t□上に文字列をコピーして、それへのポインタを返す。

・Nnum(nbuf)
nbufには文字列の形で格納された数値をとる。その数値を表すterm型構造体を作って返す。

・Nnum_val(x)
xには浮動小数による数値を取る。xを表すterm型構造体を作って返す。

・Nstr(x, flag)
xに文字列、flagには{STING.ETENAL,TEMPORAL}を取る(nalloc参照)。xを表すterm型構造体を作って返す。

・Nvar(nbuf)
新たなvarを定義する。nbufに変数名を入れて呼ぶ。大域変数v_numberには変数番号、v_listに変数リストのトップが入っているとする。Nvar()の中では、v_number,v_listが書き換えられるので、連続してvarを定義するときは、一番最初にv_number=0,v_ list=NULLとしてから連続して呼べばよい。新しくできたvar変数のアドレスを返す。

・varsearch(varname)
大域変数v_listで表される変数リストの中でvarnameという名前のものを返す。

・reset_voccurence(vl)
vlのv_occurenceをすべて0にする。

・recalc_voccur_sub(t,vl)
tを項、vlを変数リストにとる。vlのv_occurenceを再計算する。

・recalc_voccurence(cl,vl)

c1を節の本体、v1を変数リストにとる。v1のv_occurenceを再計算する。新述語定義および簡約化で用いている。

・exist_fname(fname)
fnameという名前(引数の個数は問わない)の述語が存在する場合に、その述語を返す。存在しない場合にはNULLを返す。

・predicate(fname, arity)
述語名fname、アリティarityの述語を返す。ハッシュテーブルに述語が登録されていない場合には、新たにユーザ定義述語として登録する。

・funcsearch(fname, arity)
fnameには述語名、arityには引数の個数が入る。ハッシュテーブルを調べて対応する述語のポインタ(func構造体)を返す。arity==-1の時は、fnameという述語のポインタを一つ返す。fnameで示される述語がないときには、NULLを返す。

・pred_compare(f1,f2)
述語f1,f2を比べて、f1<f2ならば-1, f1=f2ならば0、f1>f2ならば1を返す。比較は次のように行う。
(1)同一関数名でないならば、strcmp(f1->f_name,f2->f_name)
(2)同一関数名のときは引数の数の小さい方が大きい
(3)同一関数名で引数の数が同じならば、=

・void index_func(f,n,a)
fには述語のポインタ、nには述語名、aには引数の個数が入る。述語をハッシュテーブルに登録する。

・void index_funclist(f)
fは述語リストが入る。f中の述語を全てハッシュテーブルに登録する。

・Nfunc(ftype, n,a)
ftypeには述語のタイプ(TEMPFUN or not)、nには述語名、aには引数の個数を入れて呼ぶ。新述語を作り、そのfunc構造体のポインタを返す。ftype!=TEMPFUNの時は述語はハッシュテーブルに登録され、ftype==TEMPFUNの時(例えば制約変換の新述語)は述語は大域変数f_listではじまる述語リストに登録される。

・add_f_cbind(t)
tを頭部の項とする。述語f (=t->type.t_func)のf_cbind[i](i番目の引数が何回定数にバインドするか)に加える。

・recalc_f_cbind(f)
述語fのf_cbind[i]を再計算する。

・void recalc_pred_value(f)
簡約やretractにより述語定義の一部が変化したルーチンの最後に呼ぶ。f->f_cbind□やf->f_setcount,f->f_unitcountの値を再計算する。

・Nterm(n)

n引数のterm構造体をsheap上にallocする.

・Ntempterm(n)

n引数のterm構造体をuheap上にallocする.

・Nenv(n)

nには整数(変数の個数)を入れて呼ぶ。変数の数に対応する環境をユーザーヒープ上にとる。pair構造体は2つのポインタのペアなので、MS-DOSラージモデルの場合は4ワード(=8バイト)、そのほかの場合は2ワード必要となる。

・Nnode(l)

refuteで用いる。lにはノードへのポインタを入れて呼ぶ。
n->n_link=lとなるようなノードnをユーザーヒープ上に作り、そのアドレスを返す。n->n_lastはcutfが定義されているときはl,それ以外はグローバル変数のn_lastになる。n_last,cutfが書き換えられて処理が終わる。

・Ncnode(l)

制約変換で用いる。lにはcnodeへのポインタを入れて呼ぶ。
n->n_last=lとなるようなノードnをユーザーヒープ上に作り、そのアドレスを返す。

・Neclause(c,e,ec)

cを節、eをその環境、ecを任意のeclauseとして呼ぶ。(c,e)と等価なeclauseをつくりその末尾にecをつけて返す。つまり、

c=c1,c2,c3

の場合、

(c1,e)-(c2,e)-(c3,e)-ec

なるeclauseリストを返す。

・Nallvar(vl,e)

vlを変数リスト、eをその環境として呼ぶ。(vl,e)の中の具体化していない変数に対応するallvar構造リストを作り、返す。

・struct set *setconcat(slist, s)

slistは定義の列、sを定義とする。slistの最後にsをくっつけて返す。

・void add_set(s,flag)

sを定義にとる。sを述語定義に加える。flagが'a'のときは定義の先頭に、'z'の時は定義の最後に加わる。

・traceappend(t,c)

cは元の制約節、tにはintegrate()で変換されたcと等価な項、を入れて呼ぶ。cの中の項の個数、変数の個数を計算し、newf_ listに付け加える。
例えば、

t=c0(X,Y,Z)
c=append(X,Y,Z),memb(X,Z)

だと、it_cnumber=2、it_vnumber=3になる。

・upush(p)

pにはアドレスを入れて呼ぶ。pのアドレスとその中身をペアにして

スタックに格納する。MS-DOSラージモデルでは、アドレスが32ビット
になるので2回に分けて格納している。STACKBOTTOMはスタック領域の最小アド
レス。STACKTOPは最大アドレスである。

・undo(u)

uには古いスタックポインタを入れる。uspがuになるまでスタッ
クをポップしスタックに積まれていたアドレスを元に戻す。

・show_heap_max()

デバッグ用%Mコマンド。heap, ustackの最大使用量を表示

☆ read.c ドキュメンテーション
cu-Prolog

第1版　88.7.3
第2版　89.6.28
89.11.7 (for ver2.4)
90.3.14 (ver.3.0) 90.7.13(ver3.02)

§1.概要

　　このモジュールはホーン節の読み込みに関する関数の集まりである。

§2.マクロの説明

#define numeric(x)
文字xが数値または'-'の時にtrue.

#define is_varname(x)
文字xが大文字または'_'のときにtrue.読み込んだ文字列が変数かどうか
のチェックに使う。

§3.関数の説明

・next()
　　fp(入力ファイルポインタ)から1文字取り出してcbufにしまう。EOFの場合はfpのエラーをクリアーした後に,cbuf=EOFとする。

・adv()
　　空白文字をとばして、次の文字をcbufにいれて返す。

・check(c)
　　cbufがcのときはcbufに次の文字を入れて1を返す。そうでなければ0を返す。

・skip(c)
　　cbufがcでないときはエラー。そうでないときはcbufに次の文字を入れて返す。

・period()
　　cbufが"."でないときはエラー。そうでないときは何もしない。

read_first_char()
　　ユーザが入力(CRまで)した文字列の最初の文字を返す。

・keyread(a)
　　ユーザー(stdin)からの入力(文字+CR)を待つ。それがaで表される文字から始まれば1を、そうでなければ0を返す。ステップや別解探索のときに'c'を入力したかどうか調べるのに使う。

alldigit(c)
　　文字列cが全て数から成っていればtrue

void read_hexa()
　　16進数を読み込む

・Rname()
　　fpから文字列を読み込んでnbufにしまい、文字列のタイプ
(NAME,QUOTE_NAME,STRING,NUMBER,VARNAME,FILE_TYPE)を返す。
'からはじまる文字列はQUOTE_NAME
"からはじまる文字列はSTRING
#からはじまる16進文字列はFILE_TYPE
である。

・Rlist()
　　fpからリストを読み込み、そのterm構造体のアドレスを返す。
リストはDEC-10prologの記法に従い、
　　[a,b,c]は[a,[b,[c,□]]]
　　[a,b,X]は[a,[b,[X,□]]]
　　[a,b|X]は[a,[b,X]]
の略記とする。

・#define ARGMAX　　63
　　ひとつのファンクタまたは述語の引数の個数の最大値

#define CONSTANT_TERM 1
#define NOT_CONSTANT_TERM 0

・Rterm()
　　fpから項を読み込み、そのアドレスを返す。
　　項::=変数|定数|複合項
　　複合項::=述語名(項の列)
なお,定数複合項(変数を含まない複合項)を導入したために,Rterm()およびRlist()は,書き換えた．定数複合項は t->t_arityを負の値(引数の個数の符号を負にしたもの)とする．制約変換ルーチンでホーン節のコピーを大量に作る時に,定数複合項はコピーしなくて済むのでメモリの節約になる．

・Rform()
　　fpから式を読み込む。式とは変数でない項である。

Rhead()
　　ヘッドに対応する素式を読み込む。

・Rclause()
　　fpから節を読み込む。カットだけ別に扱っている。

・Rconstraint()
　　制約読み込みルーチン

・Rliteral()

　　Rconstraint()から呼ばれ,制約の1つのリテラルを読み込む

・Rvar()

　　Rliteral()から呼ばれ,制約の引数(変数)を読み込む

以上

☆ print.c ドキュメンテーション
cu-Prolog

第1版 88.7.3
第2版 89.6.28
90.3.14 ver3.0

§1.概要

このモジュールは、節や項などをプリントアウトするのに必要な関数の集まりである。

§2.変数の説明

Listlevel
リストの深さ。[a,b,□]を[a,b]と表示したいときに使っている。

§3.関数の説明

・quote_needed(f)
fの述語名を表示する時に'でくくるかどうかチェックする。アルファベット以外の文字を含んでいる時には'でくくることにする。

・tabp(n)
n個のタブをプリントする

・Pvar(t,n)
tには変数(を表すterm)、nには数を入れて呼ぶ。"変数名_ n"のようにプリントされる。ただしtが無名変数(_)の場合は"_n"とプリントされる。nが-1のときはtの変数名のみをプリントする。

・Pterm(t,e)
tを項、eをその環境として呼び、eのもとでのtを表示する。tそのものをプリントしたいときはe=NULLとして呼ぶ。リストだけ、特別なプリントルーチンがいる。

・Plist(t,e)
tをリスト項、eをその環境として呼ぶ。現在、次のような記法を許している。
[a,[b,[c,□]]]->[a,b,c]
[a,[b,X]]->[a,b|X]

・Pclause(c,e)
環境e付き節cの表示を行う。Pterm()を使っている。

・Peclause(c)
eclauseのcの表示を行う。

・Pallvar(a)

allvar構造aの表示を行う。

・Showhorn(c,cst,e)

cを定義節、cstを制約,eをそれらの環境とすると、(c,e)をA:-B;C.の形に表示する。c=c1,c2,c3……ならば、c1:-c2, c3…….となる

・Pgoal(n)

nはノード。refute()でトレース時に、制約付ゴールを表示する。

・Pconstraint(co)

coはconstraint。制約を表示する。

・Showfunc(f)

述語fの定義を、すべてプリントする。

・Showtrace(c)

cをintegrate()のトレースの定義節として、定義をA<=>B. の形で表示する。c=c1,c2,c3……ならば、

c1<=>c2,c3…….

となる

・Shownewfunc()

integrateのトレースを全て表示する。cu-Prologトップレベルの%コマンドの処理。

☆ refute.c ドキュメンテーション
cu-Prolog

第3版 90.3.14 (ver.3.0)
90.7.13 (ver.3.02)

§1.概要

このモジュールはprologの反駁処理を行う。

§2.マクロ説明

ノードは一つのゴール(リテラル)に対応し、次のnode構造体で表される。

```
struct node {
        struct clause *n_clause; ゴールリテラル
        struct pair *n_env; ゴール中の変数の環境
        struct set *n_set;  ゴールリテラルに適用できる(同じ述語名を頭
                                部に持つ)節
        struct node *n_link, 親ノード
                    *n_last; バックトラック先ノード
        struct constraint *n_constraint; ゴールに対する制約
        int *n_hp, ノード作成時のユーザヒープポインタ
            n_count, ノード番号
            n_scount, ゴールの展開の回数
            n_spy, ゴールがスパイされているとき1、そうでないとき0
            n_tmp; システム述語のための一時変数
        struct ustack *n_usp; ノード作成時のユーザスタックポインタ
};
```

・is_dead(n)

deadノードとはゴールに対して適用するプログラム節がないノードをいう。

・is_tip(n)

tip(末端)ノードとは、ゴールが空のノードをいう。

・is_root(n)

rootノードとは親ノードが空のノードをいう。

・木の探索の方向を示すフラグ(include.hで定義される)

DOWN 下向き
UP 上向き
BACKTRACK バックトラック時

§3.関数説明

・struct node *Newnode(goal, icons, env, nlink, nlast)

新ノードを作る。ただし、goalはゴール、iconsは制約、envは環境、nlinkは親ノード、nlastはバックトラックノードとする。

・struct set *init_set(n)
ノードnのゴールの初期適用節(set構造体)を返す。

・struct node *backtrack_node(n)
ノードnにバックトラックする。つまり、nからn_lastをさかのぼって、deadでない(適用節を持っている)ノードをバックトラックノードとして返す。

・int have_nextgoal(n)
ノードnが次のゴールリテラルを持っているときTRUE、持っていない時FALSEを返す。

・struct node *next_goal(m,oldnode,btnode)
AND方向に新ノードを作成する。oldnodeは一番最近にできた空ゴールのノード、btnodeはバックトラックノードとして、mの次のリテラルをゴールとする新ノードを作り返す。

・struct node *proceed_node(n, btnode)
空節ノードから親をたどって、次のゴールを持っている節を見つけ、そのゴールに対する新ノードを返す。

・refute(goal, iconstraint, e, vlist)
refutation本体。goalはゴール、iconstraintは初期制約、eは初期環境、vlistは変数リスト。

※※ver3.02でrefute()は大きく書き換えられた。
refute(Root, n, Status)
main.cのquestionclause()で前処理を行なったので、こちらでは反駁処理のみを行なっている。ゴールノードnを再左導出、深さ優先探索で解いて、解がある場合にはTRUE、ない場合にはFALSEを返す。

構造をフローチャートで示すと次のようになる。

初期ノード設定
 |
(※)
m <- nを展開したもの
 |
nがdeadでないときLast_BT=nとする
 |
<展開失敗(m==NULL)か?->
 |　　<n:ルートか?->
 |　　 |　　[終了]
 |　　nをバックトラックノード、環境を整え(※)へ
 |
<mは末端ノードか?->
 |　　<次のゴールを持つノードがあるか?->
 |　　 |　　AND方向に新ノードnを作り(※)へ
 |　　解を表示

```
|     次の解を探す場合はLast_BTにバックトラックして(※)へ
|     次の解を探さない場合は[終了]
|
```

n=mとして(※)へ

・extend(n,status,btnode)
ノードnを展開して、子供ノードを作って返す。
すなわち
1.n->n_setよりゴールに適用する節sを一つ除く
2.nのゴールを節sで展開する。
3.sの本体と2における環境をもった節mを作る。
4.mを返す。
システムの組込み述語に対しては
(1) 関数的な組込み述語(常に一つの解しかない)の場合
　　system_function()関数
(2) 非関数的な組込み述語(複数の解を持つ)の場合
　　system_pred()関数
により実行する。

・resolve(n0, n, sliteral, env)
n0を親ノード、nを子ノード、(sliteral, env)をゴールリテラルとする。
ゴールリテラルをn0のプログラム節のひとつの頭部と単一化する。単一化子は、
n->n_envである。

・int Panswer(root, vlist, lastbt)
解を表示する。さらに解を計算するとき(ユーザが;を入力した場合)はTRUE、
そうでない時はFALSEを返す。

・void Pbinding(vlist, env)
変数の束縛を表示する。

・int Trace_Goal(n)
ノードnのゴールを表示する。ステップトレース時にrefutationを打ち切る場
合(ユーザがcを入力した場合)はTRUE、そうでない時はFALSEを返す。

・void Trace_True(n)
ゴールのrefutationが成功した場合のトレース表示

・void Trace_False(n)
ゴールのrefutationが失敗した場合のトレース表示

・void Trace_Unification(n)
ゴールと節のユニフィケーションが成功した場合のトレース表示

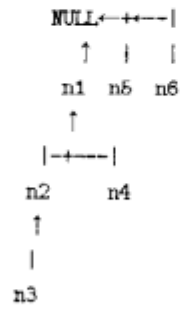
・void Trace_Answer(n)
得られた解により具体化されたゴールの表示

・void end_refute(n) (※ver3.02にて消去)
refuteルーチンを抜ける時に表示する。

§4.データ構造説明。

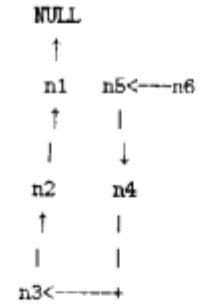
n->n_link、n->n_last
一つのノードは、2つの他ノードへのリンクをもつ。n_linkには親ノード、
n_lastにはバックトラックノードが入る。

例えばn->n_linkは



と、親子関係(向きは子から親へ)を表す。ここで深さ優先探索によりノードは
n1,n2...の順に作成される。ノードとn_linkによりAND木が表現される。
n1,n5,n6はルートノード(is_root(N)を満たす)、n3,n4は末端ノード
(is_tip(N)を満たす)である。refute()では、変数Rootはn1を指す。

上の木において n->n_last は



のようにバックトラック先ノードを表す。

○カットの処理は以下のように行う。
1. 一回目の実行は成功する。ただし、その際に親ノードの適用プログラム節(n_set)を
空にする(OR方向のカット)
2. バックトラックによる再実行は失敗する。
これらの処理はdefsysp.cのcut_pred()で行っている。

◎トレースについて
トレースは
1. トレースしたい述語にスパイフラグを立てる。
2. トレース(ノーマル/ステップ)スイッチをon/offする。
ノーマルトレースでは、途中で表示をストップしない。また
ステップトレースではゴール表示後一旦入力待ちの状態になり
リターン : continue

s [RT] : スキップ(ゴールリテラルが成功/失敗するまで表示をしない)
z [RT] : アボート(refutation自体を終了する)
を選択する。

・ノードnのゴールがスパイされている場合には
n->n_spy = 0
スパイされていない場合には
n->n_spy = 1
である。これらはinit_set()にて設定している。

・ステップトレースについては、大域変数Last_SKIPにスキップのかかったノードが格納され、それよりも下のノードについてはトレース表示をカットしている。
以上

☆ システム組込み述語(defsysp.c, syspred1.c syspred2.c)
　　ドキュメンテーション

第1版　88.10.30,11.7
第2版　89.7.26
89.11.7 (Ver.2.4)
第3版　90.3.26 (ver3.0)　　ver3.02 90.7.13

§1. 概要

　　これらのモジュールはcu-Prologの組み込み述語を定義する。主にrefute.cから呼ばれる。

　　defsysp.cでは組込み述語のエントリを定義
　　syspred1.cではassertなど一般的な組込み述語の処理
　　syspred2.cでは数値、文字列演算の組込み述語の処理
　　jpsgsub.cではjpsgパーザ用の組込み述語の処理

を行っている。

§2. 組込み述語の拡張方法

まず、cu-Prologの組込み述語には、
・タイプ1:関数的(常に一つの解を持つ:バックトラックをしない)
・タイプ2:述語的(複数の解を持つ:バックトラックする)
の2種類がある。両者で若干定義方法が異なる。

(1) 述語を表すfunc変数名を決める。これはASSERT_PやABOLISH_Pのように、基本的には述語名に_Pを付加したものにする。また、述語の処理を行う関数名を決める。これはassert_predのように基本的に_predを付加した名前にする。
述語を表す変数はsyspdef.hに、処理関数はfunclist.hに定義する。
他のモジュールからも参照される述語については述語変数をsysp.hにも定義する。

(2) defsysp.cのdefsyspred()で述語を定義する。
タイプ1の述語は
　　Def1(述語変数名, 述語名, アリティ, 処理関数名)
タイプ2の述語は
　　Def2(述語変数名, 述語名, アリティ, 処理関数名)
で定義する。例えば、

```
Def1(ABOLISH_P,"abolish",2,abolish_pred);
Def2(CUT_P, "!", 0, cut_pred);
```

である。組込みのファンクタについては処理関数名をNULLで指定する。
modularizeやlistのように、読み込みルーチンでも用いられない特別な述語については、

```
Deftemp(MODULAR_P,"modularize",2);
```

のように指定する。この場合、modularize/2という述語は述語のハッシュテーブルには登録されない。

(3) 処理関数の本体を記述する(syspred1.c, syspred2.c, jpsgsub.c等)。
(3-1)タイプ1の述語の処理
以下のような関数呼び出しにする。

```
int fail_pred(t,e,n)
struct term *t;
struct pair *e;
struct node *n;
{
        .........
}
```

(t,e)で組込み述語を含んだリテラルを表し、nはそのリテラルをゴールとするノードである。構造体nのメンバ(n->n_setなど)を無暗に書き換えてはいけない。リテラルの実行が成功した場合はSYSTRUE、失敗した場合はSYSFAILを返す。

(3-2)タイプ2の述語の処理
以下のような関数呼び出しにする。

```
int cut_pred(t,e,n,m,status)
struct term *t;
struct pair *e;
struct node *n,*m;
int status;
{
        .........
}
```

(t,e)で組込み述語を含んだリテラルを表し、nはそのリテラルをゴールとするノード、mはnの親ノード。statusはリテラルが呼ばれた時の探索の状態(DOWN,UP,BACKTRACK)を表す。
(status != BACKTRACK) の時は最初の実行、
(status == BACKTRACK) の時は再実行、となる。
n->n_tmpおよびn->n_setの値をレジスタとして使うことができる(ただし、n->n_setの値をNULLにすると、再実行時に別の値になっている恐れがあるので注意する)。リテラルの実行が成功した場合はSYSTRUE、失敗した場合はSYSFAILを返す。

§3. 変数の説明

組み込み関数の名前は、func構造体へのポインタで、MODULAR_P, INTEG_Pのように最後に_Pをつける。なお、組込み関数を他のモジュールから参照する場合には、sysp.hにも同じ名前で定義をする。

現在サポートされている(か、その作業中の)関数は、

```
write:  write(T)        項Tをコンソールに表示する
read/1: read(T)         標準(ストリーム)入力から項Tを読み込む。(未)
read/2
see/1:  see(F) ファイルFをcurrent input streamにする
seen/0: current input streamを閉じる
tab/1 :
tell/1: tell(F) ファイルFをcurrent output streamにする
told/0: current output streamを閉じる
true/0: 常に成功する述語
write/2
var:    var(T)          Tが具体化されていない変数の場合TRUE
```

univ(=..) 関数。: ml(T,L) 複合項Tをリスト形式Lにする。
name: name(X,L) 項Xの文字列のキャラクタがリストL
close/1: close(F) I/Oのために開いているファイルFを閉じる
condname/2: condname([c0(X,Y),c1(Y,Z)],X) => X=[c0,c1]
count: count(C) 呼び出し毎に異なった数を返す
end_of_file/0
gensym: gensym(N) 呼び出し毎に異なった名前を返す
nl: nl 改行
nl/1:
tab: tab タブ
pcon: pcon その時点での制約を表示する
equal: equal(X,Y) XとYをユニファイする。
is: is(X,Y) is(N,N+1)のように使う。
sum: sum(X,Y,Z) X+Y=Z (引数のうち2つ以上具体化されているときのみ)
multiply: multiply(X,Y,Z) X*Y=Z (同上)
tree: tree(Hist) ヒストリを木の形で表示する。
assert: assert(H), asserta(H), assertz(H)
retract: retract(H)
functor: functor(H,F,A) 素式Hの述語名をF、引数の数をAとする。
clause : clause(T,B,C) 素式Tと単一化する節の本体がB、制約がCとする。
(backtrackで複数の解を探索可)
open/3
or/2
or/3
or/4
or/5

なお、Ver2.4から、素式そのものを変数として記述できるようになった。従って
call(X):-X.
not(X) :- X,!,fail.
not(_).
により、call/1, not/1が定義できる。

他に定義されているfunc変数
modular_p: modularize() 内部関数(プログラム中では使えない)
integrate_p: integrate() 内部関数(同上)
cat_p: カテゴリー(単一化文法のノード)
cname_p: 制約リスト

他の変数
int COUNTNUMBER
Gensym()関数の種。

§4. 関数の説明

§4-1. defsysp.c

・defsyspred()
組み込み述語定義。main.cのprepare()関数でシステム立ち上げ時に1度だけ呼ばれる。述語名を定義している。

・system_function(t,e,n)
(t,e)で組込み述語を含んだゴールリテラル、nはノードを取る。
(t,e)が組み込み述語でないとき、 SYSNO
(t,e)が組み込み述語で、trueのとき SYSTRUE
(t,e)が組み込み述語で、failのとき SYSFAIL
を返す。

・system_pred(t,e,n,status)
(t,e)で組込み述語を含んだゴールリテラル、nをノード、statusは
UP,DOWN,BACKTRACKのいずれかを取る。
(t,e)が組み込み述語でないとき、 SYSNO
(t,e)が組み込み述語で、trueのとき SYSTRUE
(t,e)が組み込み述語で、failのとき SYSFAIL
を返す。

・cut_pred(t,e,n,status) : カット (!) の処理。
大域変数Last_BTを書き換えている。
・fail_pred(t,e,n) : fail述語(常に失敗)の処理。
・halt_pred(t,e,n) : halt述語(Prolog終了)の処理。
・abomb_pred(t,e,n) : atomicbomb述語(強制終了)の処理。
・true_pred(t,e,n,m,status) : true(常に成功)の処理

§4-2. syspred1.c
・memb_pred(t,e,n,status) : 組込み版member述語
・or_pred(t,e,n,m,status) : or/1,
・rev_clause(c) : clauseの順番を逆にする
・read_pred: read/1,2
#define SPECIFIED
#define INPUT
#define OUTPUT
ファイルのタイプ
・open_pred(t,e,n)
・see_pred(t,e,n)
・tell_pred(t,e,n)
・file_open_pred(t,e,openmode)
・seen_pred(t,e,n)
・told_pred(t,e,n)
・close_pred(t,e,n)
・pcon_pred(t,e,n) : pcon(その時点での制約を返す)述語
・cunify_pred(t,e,n) : unify(制約を変換する)述語
・write_pred(t,e,n) : write(項を表示する)述語。項がストリングの場合は
ダブルクォーテーションを表示しない。
・nl_pred(t,e,n) : CR (\n)
・tab_pred(t,e,n) : TAB (\t)
・var_pred(t,e,n) : 引数がfree varの時true
・equal_pred(t,e,n)
equal(X,Y)述語処理部分。第1引数と第2引数が、
単一化する場合はSYSTRUE
単一化失敗の場合はSYSFAIL

を返す。

- eq_pred(t,e,n)
- eq_pred_sub(x,y,ex,ey)
- int equalpred(t1,e1,t2,e2)
- int assertz_pred(t,e,flag) : assertz述語エントリ
- int assert_pred(t,e,flag) : assert,asserta述語エントリ
- void general_assert(t,e,flag) : flagは'a'又は'z'
- struct clause *list_to_clause(t,e):
- void index_set(chead, usave, const, flag):
- int retract_pred(t,e,n): retract述語
- int clear_predicate(f): clear述語
- int abolish_pred(f): abolish/2述語

int makelist_pred(t,e,n): univ述語
struct func *constsearch(fname)
　　fnameで示される定数(アトム)のポインタを返す。

int Llevel(t,e)
　　makelist()で呼ばれる。リストのレベルを返す。

void LtoP(t,tt,e)
void PtoL(t,e)
　　makelist()で呼ばれる。リスト<-->述語。

int name_pred(t,e)
　　name()述語。第1引数はstring、第2引数はそのリスト表現。

void LtoC(t,e,pos)
void CtoL(pos)
　　name()で呼ばれる。文字列<-->リスト
int arg_pred(t,e,n)
int functor_pred(t,e,n)
　　functor述語。functor(T,F,N)において、項Tのファンクタ(述語)名がF、
引数の数がN。

int make_func(f,a,t,e)
int match_func(t,e,f,ef,a,ea)
　　functor_pred()で用いられる。

int clause_pred(t,e,n) : clause述語
struct term *Clause_to_List(c): clause_predで用いられる。

§4.3 syspred2.c

int sum_pred(t,e,n)
int multiply_pred(t,e,n)

int calc_pred(t,e,op)
　　数値計算用述語。op = "+" or "*"。数値は文字列(アトム)として
取り扱っている。

```
int calc_1(x,e0,y,e1,z,e2,op)
int calc_2(x,e0,z,e2,y,e1,op)
        数値計算用サブモジュール
int greater_pred(t,e,n)
int less_pred(t,e,n)
int geq_pred(t,e,n)
int leq_pred(t,e,n)
int compare_pred(t,e,n)
int concat_pred(t,e,n)
int app_str(x,y,z,ez)
int diff_str(x,z,y,e,flist)

int concat2_presd(t,e,n)
        count述語。COUNTNUMBERを種として、呼ぶ度に1ずつ増えた数を返す。

int gensym_pred(t,e)
        Gensym述語。genfunc()を使い、呼ばれる度に異なった述語名を返す。
```

☆ JPSGSUB.c ドキュメンテーション
cu-Prolog

90.3.26 (ver.3.0)

§1. 概要

本モジュールは、jpsgパーザ用の組込み述語等を定義している。

§2. 関数説明

#define TREEMAX 20

tree()述語でプリントできる木の深さの最大値。

treehist[TREEMAX]

treeprint()用ワーク配列。

int cattype[i] : i番目の素性のタイプを表す
int catname[i][5] : i番目の素性名(大文字で始まる5文字程度)
#define Normal, CatSingle, CatSet

素性のタイプを表す。CatSingleはカテゴリのsingleton setを取る素性、CatSetはカテゴリのsetを取る(例えばSubcat)素性、Normalはそれら以外である。

以下、Ptree()からPsccontsub()までは、現在サポートしている、履歴表示関数tree()の定義である。

・show_category()

デバッグ用。catファンクタの引数タイプを表示する。

・init_category()

catファンクタを以下のデフォルトに設定。

	素性名	素性タイプ	対応するJPSGの素性
1	POS	Normal	pos
2	FORM	Normal	gr, vform, pform, etc.
3	AJA	CatSingle	ajacent
4	AJN	CatSingle	ajunct
5	SC	CatSet	subcat
6	SEM	Normal	sem

・list_to_cat(t,n)
tを%Cコマンドで読み込んだリスト項、nをtに含まれる素性の数とする。tを分解してcattype□,catname□を作成する。

・set_category()
%Cコマンドを処理する。トップレベルから、
%C [AAA,1,BBB,2,CCC,3]
のように素性名と素性タイプとをリストにして入力する。

・tree_pred(t,e,n)

組込み述語tree()エントリ。

・Ptree(t,e)
tree(H)述語処理部分。ヒストリーの定義は、次である。

(1)カテゴリーはヒストリー

(2)C,Wがカテゴリーのとき、t(C,W,□)はヒストリー

(3)L,Rがヒストリー、Mがカテゴリーまたはt(C,W,□)の形(C,Wはカテゴリー)のとき、t(M,L,R)はヒストリー

t(C,W,□)の形のヒストリーは、

C—W

t(M,L,R)の形のヒストリーは

```
---M
  |
  |--L
  |
  |—Rと表示される。
```

・int null_or_nil(t,e)
項(t,e)がNULLまたはNIL(□)の時TRUE、そうでないときFALSEを返す。
通常、素性値が空の素性は表示しない。

・PCat(t,e,f)
(t,e)にはカテゴリーを表す項、fには0/1が入る。カテゴリを

第1素性値[第2素性値，第3素性名:第3素性値，...]:最終素性値

の形で表示する(第4素性以下は第3素性と同様に表示される)。
fが1のときは第3素性以下を表示しない。これはSubcatの中身をプリントする時などに用いる。

・Psubcat(t,e)
(t,e)にはsubcat素性のようにリストをとる素性の値が入る。リストで格納されている素性の値をカンマで区切って並べて表示する。

☆ unify.c ドキュメンテーション
cu-Prolog

第1版 88.7.3
89.11.7 (for Ver2.4)

1.概要

このモジュールは、prologにおける通常の単一化を行う。

2.関数説明

いずれの関数も、単一化成功時にはそのままreturn、失敗時にはlongjmp()で、unifyを呼んだモジュール(例えばrefute.c)のsetjmp()文に大域脱出する。失敗時にはsetjmp関数の返値は1になる。jmp_buff変数名はfailである。(大域脱出を使うとプログラムが汚くなるので、いずれ書き換えた方がいいと思う)

```
void ocheck(p,t,e)
pair *p,*e
term *t
```

オッカーチェックを行う。pという環境が表す変数が、(t,e)という項に出現する場合は単一化失敗、出現しない場合は何もせずに返る。
オッカーチェックとは、例えばf(X,X)とf(X,g(X))とを単一化して、Xがg(g(…… と無限になってしまうのを防ぐことである。

```
void unify(t,e,u,f)
pair *e,*f
term *t,*u
```

単一化本体の処理を行う。つまり(t,e)=(u,f)となるように、環境e,fを書き換える。環境を書き換えるときには、upush()関数で前の値を保存している。これをやっておけば、unifyを呼んだ方のモジュールで、単一化環境を清算したいときにはundo()を行えばよい。通常のPrologと同じく、オカーチェックは行わない。

ちなみにprologにおける単一化とは次のように定義されている。
・変数はどれとも単一化する。
・アトム(定数)は同じアトムとのみ単一化する。
・複合項どうしは、ファンクタ(述語名)が等しく、対応する各引数がそれぞれ単一化するときのみ、単一化する。

```
void safe_unify(t,e,u,f)
```
オカーチェックつきの、普通のunificationを行う。

☆ cunify.cドキュメンテーション
cu-Prolog

第1版 88.11.7
第2版 89.6.28, 7.26 , 89.11.7(for Ver.2.4)
90.3.14 ver3.0

§1.概要

cunify.cは制約単一化関係の各種サブルーチンの寄せ集めである。modular.c、integ.c、split.c、reduce. c等からおもに呼ばれる。

§2.変数説明

```
#define VMAX 50

int varoccur[VMAX]
```

制約に現れる変数のうち出現頻度最大のものを選ぶルーチンで使う。VMAXは変数の個数の最大値、varoccur□は、各変数が何回出現しているかを要素にとる。varcheck()でセットされる。

§3.関数説明

関数の間の関係は次のようになっている。

```
clauseapp()
up()
upclause() ->varcheck()
             insert() -> greater()
unchanged()
upvar()
tabp()
goodterm()
occured() -> eq2() -> unify2()

transform() -> onestep_reduce() -> onestep_termreduce()
                                   compress_clase()
```

---

clauseapp(c1,c2)

c1,c2をeclauseへのポインタにとる。c1の最後にc2をつなげる。ただし、スタックにc1の最後のセルのポインタをpushしているので、undoすればリストは切れて元に戻る。

term up(t,e)

(t,e)はユーザーヒープ(hp□)に格納されている項。これを、システムヒープ(shp□)上に吸い上げ、新しく作られた項を返す。

この関数を呼ぶ前にはあらかじめ、(t,e)のうち、具体化されていない変数hvとその環境hveについては、hve->p_envが、あるallvar構造体avを差し、av->v_svarがシステムヒープ上の変数を指しているとする。新たな項はシ

ステムヒープ上に、それらの変数について構成される。なお,定数複合項(変数を含まない複合項: t->t_constant = 1)を導入する事で,無駄なコピーを作るのを防いでいる.

varcheck(t,e)

(t,e)は制約をあらわす基礎項。この中の変数の頻度をvaro ccur□配列にセットする。なお、変数の順番は、(t,e)のなかの変数に対応するshp□上の変数の変数番号によっている。

clause *insert(c,cl)

cが1つのclause、clがclauseの列を指すとする。c lの列は、項の評価値の大きい順にソートされているとする。cの評価値を計算し、clの中の正しい位置に挿入する。cと、clの項をはじめから比較していき、cが最初に大きくなったところに挿入している。

clause *upclause(ec)

ecはhp□内の、環境付き項(eclause型)を指す。ecで表されるclause列をshp□上に吸い上げるのだが、その際に項の評価値の大きい順にソートした節をつくる。また、副作用として、integ. c中で使われるECMAX(最大項)というグローバル変数に、ec中の最大項を入れる。

greater(t1,t2)

t1,t2を項として、両者の評価値の比較を行う。返す値は

t1=t2 のとき 0
t1>t2 のとき 1
t1<t2 のとき 2である。

比較方法は、基本的には 複合項 > 変数 > [ ] である。
・[ ]=[ ] [ ]はそれ以外の何よりも小さい。
・変数同士は、出現頻度が多いものほど大きい。出現頻度が同じ変数は=
・述語名、ファンクタ名のf_numberの値が大きい項の方が大。同じ述語名、ファンクタ名の項に対しては、引数を最初から比較して決める.

unchanged(a)

aは1つのallvar構造を指す。aの指す変数が何か(変数でもいい)に具体化されたときは0そうでないときは1を返す。

term *upvar(a)

allvar構造体aの指す変数(t,e)が最終的に指している変数(t.p)から、shp□上に新変数を作る。新変数の名前は、変数(t, p)の名前にv_number(この関数を呼ぶ前にあらかじめ値を入れておく)を続けたものになる。

int goodterm(t,e)

(t,e)は、変数を全く含んでいない項。モジュラー変換の対象にはならないが、この項に矛盾があっては変換が失敗してしまう。この関数は、(t,e)を通常のprologで実行して、trueなら1、failなら0を返す。

term *occurd(c,n)

cをclause型の節、nをcが含んでいる変数の個数とする。i ntegrate の履歴に、integrate(cc)=tがあり、cとc ccが変数の順序を除いて同一の場合は、項t(の変数をccにより並べ変えたもの)を返す。履歴になかった場合はNULL

を返す。

int eq2(c1,c2,e)

節c1,c2と、環境eをとる。*occured()関数で、履歴節と候補節が変数の付け替えをのぞいて同一か調べる。成功時には1、失敗時には0がかえる。副作用として、c1=(c2,e)となるように環境eを作る。

void unify2(t1,t2,e)

項t1,t2が変数の付け替えを除いて同一のものか、テストする。成功時は、t1=(t2,e)となるようにeが書き換えられ、失敗時は、j mp_buf変数eqfailを通じて、eq2()に大域脱出する。

struct clause *clauseset(cl,e)

制約変換用に、素式列をセットする。haepにある(cl,e)をsheap上に吸い上げる。

struct clause *clappend(c1,c2)

c1,c2は、clause型。c1の後にc2をつなげて返す。

struct constraint *transform(precoond, newc, newenv)

CAHCの制約変換処理を行う。precondは古い制約,(newc,newnev)は新しい制約で、両者の共通制約を返す。2つの制約の和を一旦sheap上に吸い上げ、startmodular()を呼んでいる。cu()でやっているのと同じ方法。制約変換失敗時は,trans_failでrefute()に大域脱出する。

struct clause *onestep_reduce(cl,env)

制約節(cl,env)を簡約する。制約に含まれる素式のうち、定義節が一つのみからなり、かつ頭部のみの節からなるものを手続きとして実行する。

void onestep_termreduce(c,e)
struct clause *compress_clause(cl)

これらは、onestep_reduce()のサブルーチンである。

☆ genfunc.c ドキュメンテーション
cu-Prolog

第1版　88.7.3
89.11.7 (ver.2.4)
90.7.13 (ver3.02)

§1.概要

　　このモジュールは制約変換で新たに作られる述語名を設定する。その他、ストリング、数値などの型変換に関する述語も定義されている。

§2.変数説明

nbuf□

　　include.h参照。カレント文字列バッファ

genname□

　　関数名の最初の文字列。main.cで定義される。初期値は"c"であるので、c0,c1……という新たな関数が作成されることになる。cu-Prologのトップレベルから、%n コマンドで書き換えることができる。

§3.関数説明

char *itoa(n,s)

　　整数nを文字列sに変換する。負の数にも対応している。

float atof(s)

　　数値を表す文字列sを、float数値に変換する。

void genfunc()

　　新関数名(genname□+番号)をnbufに入れて返す。

int chartonum(a,base)

　　数値を表す文字(0..9A..Z)に対応する数を返す。baseは底(例えば16進数)である。

long strtolong(s,base)

　　底baseの数値を表す文字列sに対応する数値(long)を返す。SUNのライブラリstrtol()と対応している。
(eg)strtolong("1f",16)—>31

☆ modular.cドキュメンテーション
cu-Prolog

第1版　88.11.7第2版　89.6.29

§1.概説

　　このモジュールは、制約変換の呼びだしを行う。integ.cの integrate()と相互呼び出しをしながら、変換を進めていく。

§2.変数の説明

#define MODULARMAX
　　ある変換において、modularize()の呼び出し回数の上限を決める。これ以上の回数を呼び出すとfailになる。無限ループの解消法。

jmp_buf moderror
　　modularize()で、例えばステップモードで強制終了するときに、大域脱出するときの変数。startmodular()でセットされ、modularize()から脱出する際に使う。

jmp_buf splitfail
　　制約を変数による同値類に分類するルーチンで、失敗したときの大域脱出変数。

§3.関数の説明

関数呼び出しの概要としては、

(1) unify()述語で実行する場合。CAHCによる実行の場合。

```
cu()->temset(),tolist()
|
startmodular()
|
modularize() <-> integrate()
```

(2)@モードで制約変換のみ行う場合。

```
modular()
|
startmodular()
|
modularize() <-> integrate()
```

eclause *clausereverse(eclist)
　　eclistは、eclauseを指す。eclistのリストを逆順にし、逆順リスト

のトップを返す。変数による同値類分類で、節の順番が逆転してしまうのを、直すのに使う。

modular(c)

cは制約を節の形でとる。cu-prologのトップレベルでの@モードの処理を行う。cは@の次に入力される節である。main.cから直接呼ばれる。制約を変換し、その結果を表示する。

struct clause *startmodualr(clist,varlist,varnumber)

モジュラー変換ルーチンのエントリー。clistは制約の節(shp□内)varlistはclistの中の変数の列(shp□内)var numberは変数の個数。変換成功時は、新たにできた制約の節(shp□内)を返す。失敗時は、MFAIL(include.h参照)を返す。やっていることは、

1.制約の変換,
2.新述語の定義の簡約化,
3.述語の有限フラグのセットである。

clause *modularize(n,modc)

変換アルゴリズムのmodularize()の処理本体。integ.cのintegrate()と互いを呼び合ながら処理を進める。nはノード。制約や変数リストなどをまとめて渡している。modcはこの関数を呼んだ回数。これがMODULARMAXを越えると強制的にfailする。変換が成功すれば、変換された節(新たにshp□に作られる)を返し、失敗の時はMFAILを返す。

この関数でやっていることは、

1.modcとMODULARMAXとの比較。MODULARMAXを越えていたら、MFAILを返して終了。

2.split()を呼び、制約を変数による同値類に分ける。splitの失敗は、splitfailなる変数を通じて大域脱出でわかる。split失敗のときは、MFAILを返して終了。

3.各同値類についてintegrate()を呼ぶ。一つでも失敗になる(FAILが返ってくる)の場合は、MFAILを返して終了。

4.integrate()の返値(項)を集めて、新たに節を作り、それを返して終了。

int cu(t,e)

組み込み述語のunify(旧制約リスト、新制約リスト)の処理を行う。(t,e)はhp□上の項であり、unify(引数1、引数2)の形をしている。引数1はリストの形の制約、引数2は変数でなければならない。組み込み述語unify()が成功するときは、1、failのときは0を返す。処理手順は次の通りである。

1.(t,e)の第1引数のリスト形式の制約を、shp□上に節の形で吸い上げる。そのときに変数も新たに定義する。なお、第1引数がリスト形式でないとき、ま

た、第2引数が変数でないときは、0を返して終了する。

2.shp□上の制約、変数、変数の個数を、startmodular()の変換ルーチンに渡す。

3.変換失敗(MFAILが返ってきた)の時は、0を返して終了。

4.変換された制約(shp□上)を、リストの形に直して、引数2とunifyして、1を返す。

term *tolist(c)

shp□上の節cの各項を要素に持つリストを作り、返す。

term *termset(t,e)

cu()の処理1(制約をshp□上に吸い上げる)を行う。(t,e)は制約を表すリテラル(項)。これをshp□上にコピーした項を返す。途中で、新変数も定義される。従って、この(項)。これをshp□上にコピーした項を返す。途中で、新変数も定義される。したがって、この関数を呼ぶ前に、v_number=0,v_list=NULLとしておかなければならない。ワークエリアとして、変数に対する環境pのp->p_env(フリーな変数は、p->p_body=NULLで、p_envの値は未定)を使っている。一度upした変数を再びupしないようにチェックしている。なお,定数複合項(変数を含まない複合項: 定数リストなど)を導入したために,無駄なコピーを作らなくなった.

☆ split.c ドキュメンテーション
cu-Prolog

第1版　88.11.8
89.11.7 (for Ver2.4)

§1.概要
制約変換ルーチン(modularize())で、素式列を、変数による同値類に分ける処理を行っている。

§2.変数の説明

extern jmp_buf splitfail
　　同値類分割に失敗したときは、大域脱出(longjmp)で、modular.cのmodularize()関数に戻る。divide()で使われる。
分割が失敗するのは、制約の中の変数を含んでいない項をprologとして実行した時にfailした場合である。

§3.関数の説明

関数呼び出しの概要(上が下を呼んでいる):

```
modularize()
|
split()
|
divide()->modularize()に大域脱出
|
avsearch()
|
contained()
|
eq()
```

eclause *ecreverse(eclist)
　　eclistは、環境付き節のリスト。eclistを逆順にして戻す。

cnode *split(n)
nは節の制約を含んだcnode。
制約を変数による同値類に分類し、cnodeのリストで返す。
例えば、制約:f(X,Y,Z),g(Z),h(U,V)　変数:X,Y,Z,U,Vが、
　　X,Y,Z .... f(X,Y,Z),g(Z)
　　U,V ..... h(U,V)
の2つのノードの列に分割される。

手順は、
1.まず、divide()で、制約の節を変数のallvar構造のv_eclauseにくっつける。

X……g(Z),f(X,Y,Z)
Y……g(Z),f(X,Y,Z)
Z……g(Z),f(X,Y,Z)
U……h(U,V)
V……h(U,V)

これは、一つ一つの素式にどの変数が含まれているか調べながら、増進的にクラスを作っていく。上の例では、f(X,Y,Z)を見て、

X...f(X,Y,Z)
Y...f(X,Y,Z)
Z...f(X,Y,Z)

ができ、次にf(Z)をみると、Zはすでに、f(X,Y,Z)がついているので、g(Z)の後に、f(X,Y,Z)をくっつける。また、Z以外の変数でf(X,Y,Z)がついているものも、g(Z),f(X,Y,Z)を指すようにする。その結果、

X...g(Z),f(X,Y,Z)
Y...g(Z),f(X,Y,Z)
Z...g(Z),f(X,Y,Z)

となる。

2.クラス毎に、新cnodeを作り、変数、制約を分割して格納する。

divide(c,a)

cには素式列としての制約。aはcに含まれる変数に対応した、al lvar構造リスト。split()で述べたような手順で、cの素式をクラス分けしながら、変数の後にくっつけていく。split()から呼ばれる。static変数SERCHEDは、avsearch()で素式がどれかの変数の後に付くと1になるので、変数を含まない素式を検出できる。そのような素式は同値類の対象にはならないが、goodterm()(cunify.c参照)により、prologとして実行して、矛盾がないか確認しなければならない。もし、矛盾があった場合は、変数splitfailを通して、modulrize()に大域脱出する。

avsearch(t,e,c,avlist)

素式を変数の後にくっつける処理を行う。divide()から呼ばれる。(t,e)は、検索中の項(cもしくはcの引数など)をあらわす。再帰的に呼ばれるたび変わる。cは対象となる素式。avlistは変数に対応するallvarリスト。

int contained(c1,c2)

c1は素式、c2は素式列を表す。c1がc2の中に含まれている場合は1、含まれていない場合は0を返す。avsearch()から呼ばれる。

int eq(t1,e1,t2,e2)

contained()から呼ばれる。(t1,e1)と(t2,e 2)とが同一の素式の場合は1、そうでないときは0を返す。

☆ reduce.c ドキュメンテーション
cu-Prolog

第1版 88.7.19
第2版 89.6.29
89.11.7 (for Ver. 2.4)

§1.概要

このモジュールは、制約変換で新たに作られた定義節の簡約を行う。現在の所は、定義ホーン節が一つしかない述語を強制的に書き換えているだけである。

§2.関数説明

Svar(v)
補助関数。変数リストvの表示をする。

copyvar(varlist)
変数リストvarlistのコピーを返す。

nthvar(varlist,n)
変数リストvarlistのn番目の変数(vnumber(v)= nとなる変数)を返す。

copyterm(oldterm,newvar)
oldtermのヴァリアントをつくる。つまり、oldtermの中の変数のみを変数リストnewvarで置き換えたコピーを返す。定数複合項(変数を含まない複合項)を導入する事で、無駄なコピーは作っていない。

copyclause(c,v)
素式列cの新変数vによるヴァリアントを返す。

remainvar(varlist,p)
varlistを変数リスト、pをその環境とする。varlistのうちinstantiateされていない変数のみをつなげて返す。

varappend(v1,v2)
v1,v2を変数リスト。v1の後にv2をつなげたものを返す。

insertcl(c1,clist,c2)
引数はいずれもclause。c1->clist->c2という構造を作り、c2の一つ前の素式(c->c_link=c2となるc)を返す。

void rename_vname(c, c)
nには変数名、nには数を入れる。cの最初のアルファベット部分に、nをつなげたものを新たな変数名とする。

varrernum(v)
変数リストvのvnumberを(0から順に)書き換える。変数名もrename_vname()を用いて [元の変数名]+[変数番号] に書き換える。

headupdate(t,p)
　　項tをその環境pによりアップデートする。

reduce(s)
　　定義ホーン節を簡約する。sに定義(set)をいれる。
例えば
　　1. f(a,V):-u(V).
　　2. g(X,Y,Z):-h(X),f(Y,Z).
とすると、節の簡約の手順は次の通り。

・1節に含まれる変数をコピーする。(V'とする)
・2.のf(Y,Z)と、1のヴァリアント f(a,V')とを単一化する。
・f(Y,Z)の代わりにu(V')を置き換える。
・2の変数のうち具体化していないもの(X,Y)と、V'とをつなげてあらたな変数リストとする。このときvnumberも新しくする。

以上により2.は
　g(X,a,V'):-h(X),u(V').
となる。

struct clause *cutclause(clist)
　　clistのうち、空素式(c_formがNULLのもの)を取り除く。

struct clause *clausereduce(clist,e)
　　素式列clistを簡約する。

struct clause *copy_append(corg, c0)

struct clause *termreduce(t,e,c)

☆ integ.cドキュメンテーション
cu-Prolog

第1版　88.11.8
89.11.7 (ver2.4)

§1.概説
　　モジュラー変換ルーチンの、integrate()の処理を行う。

§2.変数の説明

eclause *ECMAX
　　評価値の最も高い素式が入る。cunify.cのupclause()でセットされる。

§3.関数の説明
　　メインとなるのはintegrate()で、残りの関数は、integrate()から呼ばれる。

printvar(vl)
　　変数リストvlを表示する。デバッグ用。

clause *msolve(n)
　　m可解モードでは、新述語の定義節のうち一つだけを右辺をモジュラーにし、あとは右辺に依存関係があってもよいとしている。この関数は、後者の定義節の右辺を作り出す。

eclause *targetclasue(eclist)
　　eclist素式列の中で、展開すべき項を返す。選択の順位は、
　　1.有限な述語による素式
　　2.リスト(□を除く)など変数以外を引数に持つ素式
　　3.□ (NIL)を引数に持つ素式である。
これらのいずれにも当て嵌まらない場合には、NULLを返す。

eclause *removeclause(ec,eclist)
　　ecは項、eclistは節。eclistの中からecを取り除いた節を返す。

term *integrate(n,modc,intc)
　　制約変換ルーチン本体。
nは制約、変数を含むcnode,
modcはmodularizeの回数,
intcはintegrateの回数である。
　　変換が成功したときは、変換した項を返し、失敗したときは、FAIL
を返す。

主な流れとしては、次のようになる。
1.制約が□のときは、□を返す。
2.制約が一つのモジュラーな素式のときは、それをshp□にコピーしたものを返す。
Normal:

3.制約のうち、展開すべき項を選ぶ。targetclause()がNULLの場合は、ECMAX項を選ぶ。
4. 新述語名を決定する。変換がうまくいったときに、返す項(treturn)も定義する。
UNIFY:
5.制約をunfold(展開)し、modularize()に渡す。
6.新述語の定義節を付け加える。